OXFORD

Data-Driven Modeling & Scientific Computation

Methods for Complex Systems & Big Data

SECOND EDITION



J. NATHAN KUTZ

Data-Driven Modeling & Scientific Computation

Data-Driven Modeling & Scientific Computation

Methods for Complex Systems & Big Data

SECOND EDITION

J. Nathan Kutz
University of Washington

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

© J. Nathan Kutz 2026

The moral rights of the author have been asserted.

First Edition published in 2013
Second Edition published in 2026

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted, used for text and data mining, or used for training artificial intelligence, in any form or by any means, without the prior permission in writing of Oxford University Press, or as expressly permitted by law, by licence or under terms agreed with the appropriate reprographics rights organization. Enquiries concerning reproduction outside the scope of the above should be sent to the Rights Department, Oxford University Press, at the address above.

You must not circulate this work in any other form
and you must impose this same condition on any acquirer.

Links to third party websites are provided by Oxford in good faith and
for information only. Oxford disclaims any responsibility for the materials
contained in any third party website referenced in this work.

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data
Data available

Library of Congress Control Number: 2025949856

ISBN 9780198929093

ISBN 9780198929086 (pbk.)

DOI: 10.1093/oso/9780198929093.001.0001

Printed and bound by
CPI Group (UK) Ltd., Croydon, CR0 4YY

The manufacturer's authorized representative in the EU for product safety is
Oxford University Press España S.A. of Parque Empresarial San Fernando de Henares,
Avenida de Castilla, 2 – 28830 Madrid (www.oup.es/en or product.safety@oup.com).
OUP España S.A. also acts as importer into Spain of products made by the manufacturer.

For Kristy, Lauren and Katie

Contents

Acknowledgments x

0 Prolegomenon to Modern Computing 1

0.1 Perspectives from Kant and ChatGPT 3
0.2 Foundationalization of Science and Engineering, and a Warning 5
0.3 How to Use This Book 6

PART 1 **Basic Computations and Visualization**

1 Python Introduction 11

1.1 Vectors and Matrices 11
1.2 Logic, Loops, and Iterations 17
1.3 Iteration: The Newton–Raphson Method 21
1.4 Function Calls, Input/Output Interactions, and Debugging 25
1.5 Plotting and Importing/Exporting Data 29

2 Linear Systems 38

2.1 Direct Solution Methods for $\mathrm{Ax} = \mathrm{b}$ 38
2.2 Iterative Solution Methods for $\mathrm{Ax} = \mathrm{b}$ 43
2.3 Gradient (Steepest) Descent for $\mathrm{Ax} = \mathrm{b}$ 46
2.4 Eigenvalues, Eigenvectors, and Solvability 51
2.5 Eigenvalues and Eigenvectors for Face Recognition 56
2.6 Nonlinear Systems 64

3 Numerical Differentiation and Integration 69

3.1 Numerical Differentiation 69
3.2 Numerical Integration 75
3.3 Implementation of Differentiation and Integration 79
3.4 Differentiation of Noisy Data 85

4 Curve Fitting 90

4.1 Least-Square Fitting Methods 90
4.2 Polynomial Fits and Splines 95
4.3 Data Fitting with Python 99
4.4 Sparsity for Learning a Curve Fit 104

5 Basic Optimization 108

5.1 Unconstrained Optimization (Derivative-Free Methods) 109
5.2 Unconstrained Optimization (Derivative Methods) 114
5.3 Linear Programming 119
5.4 Simplex Method 124
5.5 Genetic Algorithms 128

6 Advanced Curve Fitting and Machine Learning 134

6.1 Machine Learning is Curve Fitting 134
6.2 Neural Networks as Curve Fits 139
6.3 Learned Models: Generalization, Interpolation, and Extrapolation 144

7 Visualization 152

7.1 Customizing Plots and Basic 2D Plotting 152
7.2 More 2D and 3D Plotting 160
7.3 Movies and Animations 165

PART 2 **Differential and Partial Differential Equations**

8 Initial and Boundary Value Problems of Differential Equations 171

8.1 Initial Value Problems: Euler, Runge–Kutta, and Adams Methods 172
8.2 Error Analysis for Time-Stepping Routines 179
8.3 Advanced Time-Stepping Algorithms 184
8.4 Boundary Value Problems: The Shooting Method 187
8.5 Implementation of Shooting and Convergence Studies 195
8.6 Boundary Value Problems: Direct Solve and Relaxation 199
8.7 Implementing Python for Boundary Value Problems 202
8.8 Linear Operators and Computing Spectra 207
8.9 Neural Networks for Time-Stepping 214

9 Finite Difference Methods 219

9.1 Finite Difference Discretization 219
9.2 Advanced Iterative Solution Methods for $\mathrm{Ax} = \mathrm{b}$ 225
9.3 Fast Poisson Solvers: The Fourier Transform 225
9.4 Comparison of Solution Techniques for $\mathbf{Ax} = \mathbf{b}$: Rules of Thumb 229
9.5 Overcoming Computational Difficulties 235

10 Time- and Space-Stepping Schemes: Method of Lines 240

10.1 Basic Time-Stepping Schemes 241
10.2 Time-Stepping Schemes: Explicit and Implicit Methods 245
10.3 Stability Analysis 249
10.4 Comparison of Time-Stepping Schemes 253
10.5 Operator Splitting Techniques 256
10.6 Optimizing Computational Performance: Rules of Thumb 259

11 Spectral Methods 266

11.1 Fast Fourier Transforms and Cosine/Sine Transform 267
11.2 Chebyshev Polynomials and Transform 270
11.3 Spectral Method Implementation 274
11.4 Pseudo-Spectral Techniques with Filtering 277
11.5 Boundary Conditions and the Chebyshev Transform 282
11.6 Implementing the Chebyshev Transform 286
11.7 Computing Spectra: The Floquet–Fourier–Hill Method 291

12 Finite Element Methods **298**

12.1 Finite Element Basis 298
12.2 Discretizing with Finite Elements and Boundaries 303

PART 3 **Computational Methods for Data Analysis**

13 Statistical Methods and Their Applications **311**

13.1 Basic Probability Concepts 311
13.2 Random Variables and Statistical Concepts 318
13.3 Hypothesis Testing and Statistical Significance 327

14 Time–Frequency Analysis: Fourier Transforms and Wavelets **334**

14.1 Basics of Fourier Series and the Fourier Transform 335
14.2 FFT Application: Radar Detection and Filtering 342
14.3 FFT Application: Radar Detection and Averaging 349
14.4 Time–Frequency Analysis: Windowed Fourier Transforms 353
14.5 Time–Frequency Analysis and Wavelets 360
14.6 Multi-Resolution Analysis and the Wavelet Basis 367
14.7 Spectrograms and the Gábor Transform in Python 372
14.8 Image Processing and Denoising 381
14.9 Diffusion and Image Processing 387
14.10 Compressive Sensing and Circumventing Nyquist 393

15 Matrix Decompositions **402**

15.1 The Singular Value Decomposition (SVD) 403
15.2 The SVD in Broader Context 408
15.3 Introduction to Principal Component Analysis (PCA) 414
15.4 Principal Components, Diagonalization, and SVD 418
15.5 Principal Components and Proper Orthogonal Modes 422
15.6 Robust PCA 429
15.7 Dynamic Mode Decomposition (DMD) 436
15.8 Koopman Operators 442
15.9 Randomized Linear Algebra and Scalable SVD and DMD 448
15.10 Autoencoders and Nonlinear SVD 451
15.11 Shallow Recurrent Decoder (SHRED) and Nonlinear SVD 458

16 Independent Component Analysis **462**

16.1 The Concept of Independent Components 462
16.2 Image Separation Problem 469
16.3 Image Separation and Python 475
16.4 FastICA Algorithm 480

17 Unsupervised Machine Learning **484**

17.1 Data Mining, Feature Spaces, and Clustering 484
17.2 Unsupervised Clustering Algorithms 488

18 Supervised Machine Learning **495**

18.1 Supervised Learning: Recognizing Dogs and Cats 496
18.2 The SVD and Linear Discrimination Analysis 501
18.3 Classification Trees, Support Vector Machines, and Neural Networks 508

19 Reinforcement Learning **514**

19.1 Mathematical Architecture of Reinforcement Learning 514
19.2 Markov Decision Process for Reinforcement Learning 518
19.3 Policy Optimization 520

20 Spatio-Temporal Data and Dynamics **524**

20.1 Modal Expansion Techniques for PDEs 524
20.2 PDE Dynamics in the Right (Best) Basis 528
20.3 The POD Method and Symmetries/Invariances 533
20.4 POD Using Robust PCA 540
20.5 Shallow Recurrent Decoders (SHRED) for Sensing and PDEs 544
20.6 Sparse Identification of Nonlinear Dynamics (SINDy) 549
20.7 Deep Learning Paradigms for Time–Space Stepping 554

21 Data Assimilation Methods **557**

21.1 Theory of Data Assimilation 557
21.2 Data Assimilation, Sampling, and Kalman Filtering 562
21.3 Data Assimilation for the Lorenz Equation 566

22 Transformers and Foundation Models **574**

22.1 Vector Spaces for Word Embeddings 575
22.2 The Transformer and Attention Mechanism 580
22.3 Foundation Models for Science and Engineering 585

Bibliography 587

Index 596

Acknowledgments

Acknowledgements (second edition) There are some really amazing people that I would like to thank for shaping my thinking over the last decade in computing. Foremost is my colleague Steven Brunton who has been a long-standing research partner working with me on integrating scientific computing and machine learning. His influence on my perspectives have been tremendous. I would also like to especially thank Joseph Bakarji, Urban Fasel, Bethany Lusch, Travis Askham, Jan Williams, Megan Ebers, Mars Gao, Sara Ichinaga, Sam Rudy, and Jake Stevens-Haas. These students and postdocs have been instrumental in helping me develop my thinking and algorithms in the machine learning domain presented here. Indeed, they were often my tutors on developing code for the many examples included in this book.

Acknowledgements (first edition) The idea of the first part of this book began as a series of conversations with Dave Muraki. It then grew into the primary set of notes for a scientific computing course whose ambition was to provide a truly versatile and useful course for students in the engineering, biological, and physical sciences. And over the last couple of years, the book expanded to include methods for data analysis, thus bolstering the intellectual scope of the book significantly. Unbeknownst to them, much of the data analysis portion of the book was heavily inspired by the fantastic works of Emmanuel Candés, Yannis Kevrekidis, and Clancy Rowley and various conversations I had with each of them. I have also benefitted greatly from early discussions with James Rossmanith, and with implementation ideas with Peter Blossey and Sorin Mitran; and more recently on dimensionality reduction methods with Steven Brunton, Edwin Ding, Joshua Proctor, Peter Schmid, Eli Shlizerman, Jonathan Tu, and Matthew Williams. Leslie Butson, Sarah Hewitt, and Jennifer O'Neil have been very helpful in editing the book so that it is more readable, useful, and error-free. A special thanks should also be given to all the many wonderful students who have provided so much critical commentary and vital feedback for improving the delivery, style, and correctness of the book. Of course, all errors in this book are the fault of my daughters' hamsters Fluffy and Quickles.

Prolegomenon to Modern Computing

0

0.1 Perspectives from Kant and ChatGPT 3
0.2 Foundationalization of Science and Engineering, and a Warning 5
0.3 How to Use This Book 6

Since 2013 when the first version of this book appeared, the world of computing has changed in significant and consequential ways. While the 1980s and 1990s saw the widespread adoption of computing and simulation science across all scientific and engineering disciplines, the last decade has seen the unprecedented rise and deployment of machine learning and AI algorithms. Thus, a new age and type of computing has emerged. The speed of adoption of these algorithms has been staggering, with open source code and data empowering the rapid pace of development, discovery, and design. While machine learning (statistical learning) and neural networks have been around for many decades, the former dating back to the founding of statistics as a field and the latter dating to at least the 1980s, it was the dual advancements of computing power and data generation that led to the paradigm shift enabled by the ImageNET data set in 2014. Deep learning had arrived. And thanks to the open source coding culture of the computer science community and the large technology companies (Google, Facebook, Microsoft et al.), algorithms and methods where quickly disseminated and improved to today's standards, where tasks in computer vision, language processing, and image recognition are now at human or super-human performance levels.

In 2013 when this book appeared, GitHub was sparsely populated (relative to today), open-source codes were not common, and there were no AI assistants like ChatGPT. More than that, a massive deep learning infrastructure for the simple and efficient development of neural networks and deep learning algorithms was non-existent. The tools available today profoundly change how and what can be accomplished with coding. Undoubtedly, what I write next has the potential of being outdated in a short time frame. However, I will attempt to give advice for people developing coding on how they might capitalize more fully on the modern tools available for code development. So here is some unsolicited advice:

GitHub: Everyone should have a GitHub page. This page should showcase your projects, but most importantly, make your code and data publicly available for reproducible research. I now require every one of my students in the classes that I teach to turn in their homework by

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0001

sharing it with me via GitHub. All graduate students and postdocs also need to have GitHub in order to share their research findings for their manuscripts. If you are interested in learning from this book, and coding practices in general, then step one is to set up your GitHub account if you don't have one already.

VS Code, Copilot, and AGIs: To be an efficient and pro-level coder, I'll at least highlight to you what I have observed from the best coders I know. Work with VS Code (Virtual Studio Code) as your development environment and get a subscription to Copilot (free for academics usually through institutional licenses) and some form of *artificial general intelligence* (AGI) agent, which includes ChatGPT (OpenAI), Claude (Anthropic), Gemini (Google), Llama (Meta), etc. Given the success of such *large language models*, Copilot and AGIs have revolutionized programming, providing rapid prototyping of code in an exceptionally efficient manner. Provided you ask for well-constrained tasks required of a code block, an AGI can deliver working code within seconds. The VS Code, Copilot, AGI integration absolutely transforms the efficiency and productivity of your algorithmic building. The responsible use of such AGI tools is also important to consider. Specifically, it is important to understand and test the code produced by the AGI. Intellectual property and licensing is also an ongoing discussion in the community with no clear strategies currently achieved. In general, it is always good practice to have code review sessions with others, especially more experienced colleagues.

Python: The first version of this book was written with MATLAB in mind. It is now written in Python. The shift is due almost exclusively to the advancement of deep learning code environments like pyTorch and JAX. The fact is that these neural network architectures dominate the landscape of leading algorithms in machine learning and AI. Copilot and AGIs are also trained on the representation of these codes on GitHub and Stack Overflow, for instance. As a result, the ability to clone codes and quickly develop modifications is driven by Python in the workplace. Python is also a much broader language than MATLAB and Julia. However, a note of warning is merited: Python is simply not as good as MATLAB in most scientific computing applications. Noteworthy are numerical integrators (e.g. ODE45), where Python time-steppers are actually significantly worse and even of suspect quality and capability. Similarly, some of the basic linear algebra routines like EIGS in Python are inferior to their MATLAB counterpart. Be that as it may, Python has emerged at the forefront since almost any serious code development will involve some aspect of data analysis and recourse to learning algorithms. Python allows for this to happen seamlessly.

Skills: How quickly can you find, clone, and modify to your needs existing code found on GitHub? If you are efficient in quickly understanding someone's code repository and understanding where modifications needs to be made, then you can be highly successful in modern computing and developing new applications. Coding now moves at an exceptional pace, which largely makes everyone uncomfortable. In fact, it is probably moving too fast overall. But in the data science space especially, there is only one way to keep up with what is going on, use the trifecta of VS Code, Copilot, and AGI and learn how to quickly digest and modify a cloned repository for solving your problem.

0.1 **Perspectives from Kant and ChatGPT**

With the advent of machine learning, sophisticated computing is now operating in inductive and deductive ways. Our traditional scientific computing, whereby a model (e.g. a partial differential equation) is specified as the truth for simulation is a deductive approach to the characterization, design, and control of a physical or engineering system. Specifically, physics-based models traditionally are posited from empirical observations and first-principles derivations, which include the consequences of conservation laws, physical constraints, self-consistent qualitative models, and expert knowledge. Thus, *deductive* governing equations are the starting point for computationally oriented scientific studies. The first two parts of this book largely focus on this aspect of computing. In contrast, the third part of the book focuses on data-driven models which are inherently a manifestation of the *inductive* reasoning process. Thus, models are derived directly from data, or fit directly to data, rather than being generated from an underlying principle. In modern times, induction and deduction reasoning paradigms are both highly successful, driving innovations in autonomy technologies such as robotics (primarily deductive, physics-based) and self-driving cars (primarily inductive, sensor-based). This trade-off between learning and representation paradigms has been recognized from the time of Plato (largely deductive "truths" in representation) and Aristotle (largely deriving inductive models from observations).

In terms of how inductive and deductive reasoning interact in the context of machine learning and traditional computing, we can follow the lines of thought of the great philosopher Immanuel Kant. Specifically, his opening four paragraphs of *Critique of Pure Reason*:

> That all our knowledge begins with experience there can be no doubt. For how is it possible that the faculty of cognition should be awakened into exercise otherwise than by means of objects which affect our senses, and partly of themselves produce representations, partly rouse our powers of understanding into activity, to compare to connect, or to separate these, and so to convert the raw material of our sensuous impressions into a knowledge of objects, which is called experience? In respect of time, therefore, no knowledge of ours is antecedent to experience, but begins with it.
>
> But, though all our knowledge begins with experience, it by no means follows that all arises out of experience. For, on the contrary, it is quite possible that our empirical knowledge is a compound of that which we receive through impressions, and that which the faculty of cognition supplies from itself (sensuous impressions giving merely the occasion), an addition which we cannot distinguish from the original element given by sense, untill long practice has made us attentive to, and skilful in separating it. It is, therefore, a question which requires close investigation, and not to be answered at first sight, whether there exists a knowledge altogether independent of experience, and even of all sensuous impressions? Knowledge of this kind is called a priori, in contradistinction to empirical knowledge, which has its sources a posteriori, that is, in experience.
>
> But the expression, "a priori," is not as yet definite enough adequately to indicate the whole meaning of the question above started. For, in speaking of knowledge which has its sources in experience, we are wont to say, that this or that may be known a priori, because we do not derive this knowledge immediately from experience, but from a general rule,

> which, however, we have itself borrowed from experience. Thus, if a man undermined his house, we say, "he might know a priori that it would have fallen;" that is, he needed not to have waited for the experience that it did actually fall. But still, a priori, he could not know even this much. For, that bodies are heavy, and, consequently, that they fall when their supports are taken away, must have been known to him previously, by means of experience.
>
> By the term "knowledge a priori," therefore, we shall in the sequel understand, not such as is independent of this or that kind of experience, but such as is absolutely so of all experience. Opposed to this is empirical knowledge, or that which is possible only a posteriori, that is, through experience. Knowledge a priori is either pure or impure. Pure knowledge a priori is that with which no empirical element is mixed up. For example, the proposition, "Every change has a cause," is a proposition a priori, but impure, because change is a conception which can only be derived from experience.
>
> **Immanuel Kant**, *Critique of Pure Reason* (1781) – first four paragraphs

Kant wrote this long before the invention of the computer, but his writings have direct relevance to thinking about machine learning and scientific computing today, and more broadly the two computing paradigms that now dominate across science and engineering. To better understand this contextualization in modern computing, I have asked ChatGPT to help us frame these opening four paragraphs in the parlance of machine learning models. Thus, the following prompt was used:

Can you rewrite these four paragraphs in the context of modern machine learning?

This is ChatGPT's very nice rephrasing of Kant in the context of computing:

> All our machine learning models begin with data; there can be no doubt about this. How can a model's capacity for learning be activated if not by data that impacts its sensors (or inputs) and triggers processes that organize, connect, or differentiate this data? This transformation of raw data into structured knowledge, which can be applied to new data, is what we understand as the learning phase of machine learning algorithms. In terms of temporal development, therefore, no machine learning model possesses knowledge prior to data exposure; it all starts with data input.
>
> However, while all machine learning knowledge starts with data, it doesn't necessarily arise solely from data. Indeed, it's quite feasible that the predictive capabilities of a model are a combination of the input data it receives and the inherent algorithms designed by humans (where the data merely provides the occasion for these algorithms to operate). It often becomes challenging to separate the inferences made purely from data from those that are constructed internally by the model's architecture, until extensive testing and tuning make us proficient at identifying this distinction. Consequently, it demands careful exploration to determine whether there exists knowledge within a model that is truly independent of data inputs, a type of knowledge referred to as 'a priori' knowledge in contrast to 'empirical' knowledge derived from data, or 'a posteriori'.
>
> The term "a priori," however, needs further clarification to fully convey its relevance in machine learning. While discussing knowledge derived from data, we might say a model knows something 'a priori' if it doesn't directly extract this knowledge from new data but

> from a general pattern or rule which, however, was initially learned from prior data. For example, if an algorithm predicts a system's failure, we say it 'knows a priori' that failure was imminent; that is, the model did not need to wait to observe the actual failure. But strictly speaking, even this 'a priori' knowledge was not independent of all experience, because the concept that such conditions lead to failure must have been encoded in the model through previous data.
>
> Thus, when we refer to 'knowledge a priori' in machine learning, we mean not merely knowledge that is independent of specific types of data, but knowledge that is entirely independent of any data. Opposite this is 'empirical knowledge,' or that which a model acquires only through data ('a posteriori'). 'A priori' knowledge in machine learning can be either pure or impure. Pure 'a priori' knowledge is devoid of any data-derived elements. For instance, a foundational principle such as 'every action in a system has a consequence' could be considered a 'pure a priori' proposition, though impure, because the concept of 'action' and 'consequence' in a model still relies on experiential data.

I have to say, I was rather impressed by this translation, as I don't think I could have said it any better.

0.2 **Foundationalization of Science and Engineering, and a Warning**

Predicting the future is a difficult task. However, I will venture from the perspective of 2024 to highlight some of the trends that I believe will emerge in computing for science and engineering in the upcoming years. The first trend relates to the exceptional and continued success of *large language models* (LLMs), such as GPT-4 (OpenAI), Claude (Anthropic), Gemini (Google), Llama (Meta), etc., as the engine for AGI. Foundation models are pre-trained on massive data sets using a transformer neural network architecture which relies on the so-called attention mechanism [1]. ChatGPT was the first LLM tool to go viral. Its success has prompted an LLM arms race among many major technology companies with the goal of creating the best AI agent possible for aiding us in just about any task imaginable, whether that be writing code blocks or a summary statement of a research article. Transformer architectures have more recently been used in science and engineering to build foundation models, or LLMs for specific domain applications [2]. The overall vision is that with enough training data in a specific domain, a foundation model can extract knowledge from that field, possibly with emergent capabilities, for a variety of downstream tasks. From time-series prediction [3] to weather forecasting [4], foundation models in different application areas are emerging almost weekly on the arXiv pre-print server. The claimed success of such foundation models relies on the philosophy (popularized by Richard Sutton in the "The Bitter Lesson" [5]) that with enough data, enough compute for training, and a large enough neural network, the best-performing models for a specific set of tasks can be constructed. Or an alternative to Richard Sutton's bitter lesson is the succinct statement by Ilya Sutskever, "In a nutshell, I had the realization that if you train, a large neural network on a large and a deep neural network

on a big enough dataset that specifies some complicated task that people do, such as vision, then you will succeed necessarily." Max Welling gave a nice rebuttal to this framework in "Do we still need models or just more data and compute?" This is especially relevant for science, where extrapolation is critical for building new technologies and understanding how to truly generalize our physics-based models.

In either case: foundation models will be ubiquitous and will lead to the *foundationalization* of every field of science and engineering. These pre-trained models will be remarkable black-box resources for any field of interest. In fact, it will probably be difficult eventually to build models that outperform the foundation models even if you are a leading expert in that domain. It is not clear what this means yet. In large part, engineering is concerned with two key objectives: control and design. Foundation models will almost certainly be a critical part of this process in the future. Science, however, is often focused on *discovery*, which perhaps we can superficially say is concerned with interpretability for the goal of extrapolation, or learning new physics. Foundation models can also be constructed for these goals, such as learning governing equations.

But a word of caution: *you shouldn't trust anyone!* This is a rather cynical comment. But given my personal experience (along with a great number of my colleagues) in cloning GitHub repositories from published works and finding that the results shown in the paper don't actually seem to work as claimed, even on the examples demonstrated in the paper, I stand completely by this statement. As an example, a recent paper did the very difficult job of extensively comparing different machine learning methods for solving fluid-related partial differential equations [6]. The results are quite disheartening, as it reflects a broader crisis of trust in the scientific machine learning community. This is not the place to discuss why this has happened. I only bring this to the attention of students, especially who tend to blame themselves for not being able to reproduce, or come close to reproducing, results even when they have the author's code. Chances are the code simply doesn't work as claimed. Unfortunately, one cannot trust the fame of the authors, the fame of their institution, or the citations being generated by the paper as proxies for good work (see for example [6]). And although benchmarks are emerging in almost every application area, they rely on self-reporting, which is overall a flawed premise for accurate assessment of method development.

0.3 How to Use This Book

There are a number of intended audiences for this book as is shown in the division of the book into three parts. Indeed, the various parts of the book were developed with different student audiences in mind. In what follows, the specific target audiences are highlighted along with the portions of the book appropriate for their use. Ultimately, this gives a roadmap on how to use this book for one's desired ends. All portions of the book have been heavily vetted by undergraduate and graduate students alike, thus improving the overall readability and usefulness. And in this edition, an attempt has been made to integrate machine learning tools into their appropriate chapters as a natural extension of traditional scientific computing methods.

OXFORD

Data-Driven Modeling & Scientific Computation

Methods for Complex Systems & Big Data

SECOND EDITION



J. NATHAN KUTZ

Undergraduate course in beginning scientific computing

The first part of this book reflects the topical coverage of material taught at the University of Washington campus for freshman–sophomore-level engineering and physical science students. Given that Python has become the programming language of choice in our engineering programs, the coverage begins by considering basic concepts and theoretical ideas in the context of programming language infrastructure. Thus, programming and algorithm development becomes an integral part of developing practical routines for computing, for instance, least-square fits, derivatives, or integrals. Moreover, simple things like making nice plots, generating movies, and importing/exporting data files with Python are incorporated into the pedagogical structure. Thus, Chapters 1 through 7 give a basic introduction to Python, to programming infrastructure (**if** and **for** loops), and to problem-solving development. In addition to these early chapters, differential equations are covered towards the end of the course. The ability to quickly and efficiently solve differential equations is critical for many junior- and senior-level courses in the engineering sciences such as aeronautics/astronautics (the three-body problem) or electrical engineering (circuit theory). Thus, the first part of this book has a well-defined, beginning scientific computing audience.

Graduate course in scientific computing

In addition to beginning students, there is a great deal of effort in providing an educational infrastructure and high-level overview of scientific computing methods to graduate students (or advanced undergraduates) from a broad range of scientific disciplines. A traditional graduate course in numerical analysis, while extremely relevant, often is focused on the mathematical infrastructure versus practical implementation. Further, many numerical analysis sequences are quite devoid of engineering, biological, and physical science students. The second part of this book provides a high-level introduction to the computational methods used for solving differential and partial differential equations. It certainly is the case that such systems provide an underlying theoretical framework for many applications of broad scientific interest. Thus, the key elements of finite difference, spectral and finite elements are all considered. The starting point for the graduate students is in Chapter 8 with stepping techniques for differential equations. On the one hand, one can envision skipping all of the first part of the book to begin the graduate level treatment of scientific computing. But on the other hand, the inclusion of Part I of this book allows graduate students to have a refresher section for simple things such as plotting, constructing movies, importing/exporting data, or simply reviewing programming architecture. Thus, the graduate student can use the first part of this book as a reference for their intended studies.

Graduate course in computational methods for data analysis

Parts I and II of this book offer a fairly standard treatment, although slanted heavily towards implementation here, of beginning and advanced numerical methods leading to the construction of numerical solutions of partial differential equations. In contrast, Part III of this book (Chapters 13 through 22) offers a unique perspective on data analysis methods.

Indeed, one would be hard-pressed to find such a treatment in any current textbook in the mathematical sciences. The aim of this third part is to introduce graduate students (or advanced undergraduates) in the sciences to the burgeoning field of data analysis. This area of research is expanding at an incredible pace in the sciences due to the proliferation of data collection in almost every field of science. The enormous data sets routinely encountered in the sciences now certainly give enormous incentive to their synthesis, interpretation, and conjectured meaning. This portion of the book attempts to bring together in a self-consistent fashion the key ideas from (i) statistics, (ii) time–frequency analysis, and (iii) low-dimensional reductions in order to provide meaningful insight into the data sets one is faced with in any scientific field today. This is a tremendously exciting area and much of this part of the book is driven by intuitive examples of how the three areas (i)–(iii) can be used in combination to give critical insight into the fundamental workings of various problems. As with Part II of this book, access to the introductory material in Part I allows students from various backgrounds to supplement their background where appropriate, thus making Part I an indispensable part of the overall architecture of the book.

Computational methods reference guide

In addition to its primary use as a textbook for either a graduate or undergraduate course in scientific computing or data analysis, the book also serves as a helpful reference guide. As a reference guide, its strength lies in either giving the appropriate high-level overview necessary along with key pieces of Python code for implementation, or the scientific applications portion of the book provides example ideas and techniques for solving a broad class of problems. Using either the applications or theory sections, or both in combination, provides an effective and quick way to refresh one's skills and/or analytic understanding of a solution technique. In practice, the most common comment I hear concerning this book is that students have found it useful long after their course work has been completed. I believe this is attributed to the low entry threshold for obtaining both practical theoretical knowledge and key snippets of Python code for use in developing a piece of code beyond what is covered in the text. Furthermore, since many high-level Python subroutines are introduced, a person referencing the book can be assured of exposure to techniques well beyond the simple and trivial methods that might be at first considered.

GitHub Repository

All code and data for this book can be found at the below listed GitHub repository. In addition, links to YouTube videos and playlists for each chapter and section of the book are provided on the GitHub link. Finally, homework and exercises for each chapter of the book can be found at the GitHub link. https://github.com/nathankutz/ScientificComputing

PART 1

Basic Computations and Visualization

Python Introduction

1

1.1 Vectors and Matrices 11
1.2 Logic, Loops, and Iterations 17
1.2.1 Bisection method for root solving 19
1.3 Iteration: The Newton–Raphson Method 21
1.3.1 Fixed points 22
1.3.2 The Newton–Raphson method 23
1.4 Function Calls, Input/Output Interactions, and Debugging 25
1.4.1 Input and output to Python codes 28
1.5 Plotting and Importing/Exporting Data 29
1.5.1 Subplots 33
1.5.2 Load, save, and print 35

The first five chapters of this book deal with the preliminaries necessary for manipulating data, constructing logical statements and plotting data. The remainder of the book relies heavily on proficiency with these basic skills.

1.1 Vectors and Matrices

The manipulation of matrices and vectors is of fundamental importance in Python proficiency. This section deals with the construction and manipulation of basic matrices and vectors. We begin by considering the construction of row and column vectors. Vectors are simply a subclass of matrices which have only a single column or row. A row vector can be created by executing any of the command structures

```
import numpy as np
x = [1, 3, 2]
x = np.array([1,3,2])
x = np.matrix([1,3,2])
```

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.
© J. Nathan Kutz (2026). DOI: 10.1093/oso/9780198929093.003.0002

which creates the row vector

$$\mathbf{x} = (1\ 2\ 3)\,. \tag{1.1.1}$$

Row vectors are oriented in a horizontal fashion. In contrast, column vectors are oriented in a vertical fashion and are created with either of the two following command structures:

```
x = np.matrix([[1], [3], [2]])
x = np.array([[1], [3], [2]])
```

where the square brackets indicate advancement to the next row. Either one of these creates the column vector

$$\mathbf{x} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}\,. \tag{1.1.2}$$

All row and column vectors can be created in this way.

Vectors can also be created by use of the colon. Thus, the following command line

```
x = np.arange(0,10.1,1)
```

creates a row vector which goes from 0 to 10 in steps of 1 so that

$$\mathbf{x} = (0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10)\,. \tag{1.1.3}$$

In Python, if 10 was the last value in the range, then it would go to 10 but not include 10. Thus, the reason to step to 10.1. Note that the command line

```
x = np.arange(0,11,2)
```

creates a row vector which goes from 0 to 10 in steps of 2 so that

$$\mathbf{x} = (0\ 2\ 4\ 6\ 8\ 10)\,. \tag{1.1.4}$$

Steps of integer size need not be used. This allows

```
x = np.arange(0,1.2,0.2)
```

to create a row vector which goes from 0 to 1 in steps of 0.2 so that

$$\mathbf{x} = (0\ 0.2\ 0.4\ 0.6\ 0.8\ 1)\,. \tag{1.1.5}$$

Thus the basic structure associated with the colon operator for forming vectors is as follows:

```
x = np.arange(a,b+h,h)
```

where $a =$ start value, $b =$ end value, and $h =$ step-size. One comment should be made about the relation of b and h. It is possible to choose a step-size h so that starting with a, you will not end on b. As a specific example, consider

```
x = np.arange(0,9,2)
```

This command creates the vector

$$\mathbf{x} = (0\ 2\ 4\ 6\ 8)\,. \tag{1.1.6}$$

The end value of $b = 9$ is not generated in this case because starting with $a = 0$ and taking steps of $h = 2$ is not commensurate with ending at $b = 9$. Thus the value of b can be thought of as an upper bound rather than the ending value. Finally, if no step-size h is included, it is assumed to be 1 so that

```
x = np.arange(0,5)
```

produces the vector

$$\mathbf{x} = (0\ 1\ 2\ 3\ 4)\,. \tag{1.1.7}$$

Not specifying the step-size h is particularly relevant in integer operations and loops, where it is assumed that the integers are advanced from the value a to $a + 1$ in successive iterations or loops.

Matrices are just as simple to generate. Now there are a specified number of rows (N) and columns (M). This matrix would be referred to as an $N \times M$ matrix. The 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 2 \\ 5 & 6 & 7 \\ 8 & 3 & 1 \end{pmatrix} \tag{1.1.8}$$

can be created by use of the semicolons

```
A = np.array([[1,3,2],[5,6,7],[8,3,1]])
```

or by using the return key so that

```
A = np.array([[1,3,2],
              [5,6,7],
              [8,3,1]])
```

In this case, the matrix is square with an equal number of rows and columns ($N = M$).

Accessing components of a given matrix or vector is a task that relies on knowing the row and column of a certain piece of data. The coordinate of any data item in a matrix is found from its row and column location so that the elements of a matrix $\mathbf{A}$ are given by

$$\mathbf{A}\,(i, j) \tag{1.1.9}$$

where i denotes the row and j denotes the column. To access the second row and third column of (1.1.8), which takes the value of 7, use the command

```
x = A[1,2]
```

This will return the value $x = 7$. Recall that Python starts indexing from zero, which is unlike MATLAB which starts indexing from one. To access an entire row, the use of the colon is required

```
x = A[1,:]
```

This will access the entire second row and return the row vector

$$\mathbf{x} = (5\ 6\ 7)\,. \tag{1.1.10}$$

Columns can be similarly extracted. Thus,

```
x = A[:,2]
```

will access the entire third column to produce

$$\mathbf{x} = \begin{pmatrix} 2 \\ 7 \\ 1 \end{pmatrix}\,. \tag{1.1.11}$$

The colon operator is used here in the same operational mode as that presented for vector creation. Indeed, the colon notation is one of the most powerful shorthand notations available in Python.

More sophisticated data extraction and manipulation can be performed with the aid of the colon operational structure. To show examples of these techniques, we consider the 4×4 matrix

$$\mathbf{B} = \begin{pmatrix} 1 & 7 & 9 & 2 \\ 2 & 3 & 3 & 4 \\ 5 & 0 & 2 & 6 \\ 6 & 1 & 5 & 5 \end{pmatrix} . \tag{1.1.12}$$

The command

```
x = B[1:3,1]
```

removes the second through third row of column 2. This produces the column vector

$$\mathbf{x} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} . \tag{1.1.13}$$

The command

```
x = B[3,1:]
```

removes the second through last columns of row 4. This produces the row vector

$$\mathbf{x} = (1\ 5\ 5) . \tag{1.1.14}$$

We can also remove a specified number of rows and columns. The command

```
C = B[:-1,1:4]
```

removes the first row through the next to last row along with the second through fourth columns. This then produces the matrix

$$\mathbf{C} = \begin{pmatrix} 7 & 9 & 2 \\ 3 & 3 & 4 \\ 0 & 2 & 6 \end{pmatrix} . \tag{1.1.15}$$

As a last example, we make use of the transpose symbol which turns row vectors into column vectors and vice versa. In this example, the command

```
D = ( B[0,1:4], B[0:3,2].T )
```

makes the first row of $\mathbf{D}$ the second through fourth columns of the first row of $\mathbf{B}$. The second row of $\mathbf{D}$, which is initiated with the semicolon, is made from the transpose (.') of the first three rows of the third column of $\mathbf{B}$. This produces the matrix

$$\mathbf{D} = \begin{pmatrix} 7 & 9 & 2 \\ 9 & 3 & 2 \end{pmatrix}. \tag{1.1.16}$$

An important comment about the transpose function is in order. In particular, when transposing a vector with complex numbers, the period must be put in before the ' symbol. Specifically, when considering the transpose of the column vector

$$\mathbf{x} = \begin{pmatrix} 3+2i \\ 1 \\ 8 \end{pmatrix}, \tag{1.1.17}$$

where i is the complex (imaginary) number $i = \sqrt{-1}$, the command

```
y = x.T
```

produces the row vector

$$\mathbf{y} = (3+2i \ 1 \ 8), \tag{1.1.18}$$

whereas the command

```
y = np.conj(x)
```

produces the row vector

$$\mathbf{y} = (3-2i \ 1 \ 8). \tag{1.1.19}$$

Thus, the use of the ' symbol alone also conjugates the vector, i.e. it changes the sign of the imaginary part.

As a final example of vector and/or matrix manipulation, consider the following vector

```
x = np.array([-1, 2, 3, 5, -2])
```

This produces a simple row vector with both positive and negative numbers. The following commands can be quite useful in practice for manipulating such a vector and setting, for example, threshold rules for the vector.

```
y1 = (x > 0) * 1
y2 = (x > 0) * x
y3 = (x > 0) * 3
```

These commands produce the vectors

$$\mathbf{y}_1 = (0\ 1\ 1\ 1\ 0)\ , \tag{1.1.20a}$$

$$\mathbf{y}_2 = (0\ 2\ 3\ 5\ 0)\ , \tag{1.1.20b}$$

$$\mathbf{y}_3 = (0\ 3\ 3\ 3\ 0)\ , \tag{1.1.20c}$$

respectively. This provides a very convenient way to quickly sweep through a vector or matrix while applying a logical statement.

1.2 Logic, Loops, and Iterations

The basic building blocks of any Python program, or any other mathematical software or programming languages, are **for** loops and **if** statements. They form the background for carrying out the complicated manipulations required of sophisticated and simple codes alike. This section will focus on the use of these two ubiquitous logic structures in programming.

To illustrate the use of the **for** loop structure, we consider some very basic programs, which revolve around its implementation. We begin by constructing a loop which will recursively sum a series of numbers. The basic format for such a loop is the following:

```
import numpy as np
a = 0
for j in range(0, 5):
    a = a + (j+1)
```

This program begins with the variable sum, **a**, being zero. It then proceeds to go through the **for** loop five times, i.e. the counter j takes the value of one, two, three, four, and five in succession. In the loop, the value of sum is updated by adding the current value of j, where we recall that Python starts from zero. Thus, starting with an initial value of sum equal to zero, we find that the variable a is equal to $1\,(j=1), 3\,(j=2), 6\,(j=3), 10\ (j=4)$, and $15\ (\ j{=}\ 5)$.

The default incremental increase in the counter j is 1. However, the increment can be specified as desired. The program

```
a = 0
for j in range(0,5,2):
    a = a + (j+1)
```

is similar to the previous program. But for this case, the incremental steps in j are specified to be 2. Thus, starting with an initial value of sum equal to zero, we find that the variable a is equal to $1\,(j=1), 4\,(j=3)$, and $9\,(j=5)$. And even more generally, the **for** loop counter can be simply given by a row vector. As an example, the program

```
a = 0
loop = [1,5,4]
for j in loop:
    a = a + j
```

will go through the loop three times with the values of j being 1, 5, and 4 successively. Thus, starting with an initial value of a equal to zero, we find that a is equal to 1 ($j=1$), 6 ($j=5$), and 10 ($j=4$).

The **if** statement is similarly implemented. However, unlike the **for** loop, a series of logic statements is usually considered. The basic format for this logic is as follows:

```
if (logical statement):
      (expressions to execute)
elseif (logical statement):
      (expressions to execute)
elseif (logical statement):
      (expressions to execute)
else:
      (expressions to execute)
```

In this logic structure, the last set of expressions is executed if the three previous logical statements do not hold. The basic logic architecture in Python is given in Table 1.1. Here, some of the common mathematical logic statements are given such as, for example, equal to, greater than, less than, AND, and OR.

In practice, the logical **if** may only be required to execute a command if something is true. Thus, there would be no need for the *else* logic structure. Such a logic format might be as follows:

```
if (logical statement):
      (expressions to execute)
elseif (logical statement):
      (expressions to execute)
```

In such a structure, no commands would be executed if the logical statements do not hold.

Table 1.1 Common logic expressions in Python used in **if** statement architectures.

logic	Python expression
equal to	==
not equal	~=
greater than	>
less than	<
greater than or equal to	>=
less than or equal to	<=
AND	&
OR	|

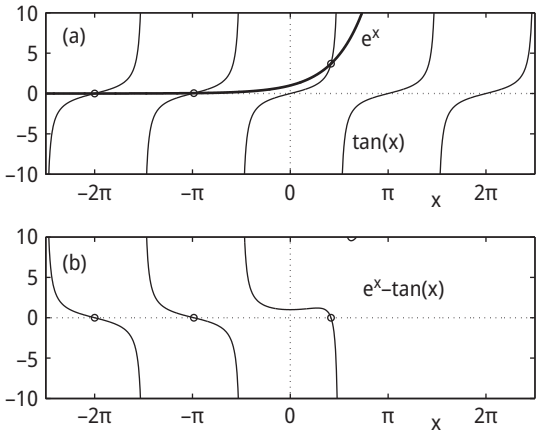


Figure 1.1 (a) Plot of the functions $f(x) = \exp(x)$ (bold) and $f(x) = \tan(x)$. The intersection points (circles) represent the roots $\exp(x) - \tan(x) = 0$. In (b), the function of interest, $f(x) = \exp(x) - \tan(x)$ is plotted with the corresponding zeros (circles).

1.2.1 Bisection method for root solving

To make a practical example of the use of **for** and **if** statements, we consider the bisection method for finding zeros of a function. In particular, we will consider the transcendental function

$$\exp(x) - \tan(x) = 0 \tag{1.2.1}$$

for which the values of x which make this true must be found computationally. Figure 1.1 plots (a) the functions $f(x) = \exp(x)$ and $f(x) = \tan(x)$ and (b) the function $f(x) = \exp(x) - \tan(x)$. The intersection points of the two functions (circles) represent the roots of the equation. We can begin to get an idea of where the relevant values of x are by plotting this function. The following Python script will plot the function over the interval $x \in [-10, 10]$.

```
import matplotlib.pyplot as plt
dx = 0.1
x = np.arange(-10,10+dx,dx)
y = np.exp(x)-np.tan(x)
plt.plot(x,y)
plt.axis([-10, 10, -10, 20])
```

It should be noted that $\tan(x)$ takes on the value of $\pm\infty$ at $\pi/2 + n\pi$, where $n = \ldots, -2, -1, 0, 1, 2, \ldots$. By zooming in to smaller values of the function, one can find that there are a large number (infinite) of roots to this equation. In particular, there is a root located at $x \in [-4, 2.8]$. At $x = -4$, the function $\exp(x) - \tan(x) > 0$, while at $x = -2.8$ the function $\exp(x) - \tan(x) < 0$. Thus, in between these points lies a root.

The bisection method simply cuts the given interval in half and determines if the root is on the left or right side of the cut. Once this is established, a new interval is chosen with the midpoint now becoming the left or right end of the new domain, depending of course on the location of the root. This method is repeated until the interval has become so small and the function considered has come to within some tolerance of zero. The following algorithm uses an outside **for** loop to continue cutting the interval in half while the embedded **if** statement determines the new interval of interest. A second **if** statement is used to ensure that once a certain tolerance has been achieved, i.e. the absolute value of the function $\exp(x) - \tan(x)$ is less than 10^{-5}, then the iteration process is stopped.

```
xr = -2.8; xl = -4
for j in range(0, 100):
    xc = (xr + xl)/2
    fc = np.exp(xc) - np.tan(xc)
    if ( fc > 0 ):
        xl = xc
    else:
        xr = xc

    if ( abs(fc) < 1e-5 ):
        display(xc); display(j)
        break
```

Note that the **break** command ejects you from the current loop. In this case, that is the j loop. This effectively stops the iteration procedure for cutting the intervals in half. Furthermore, extensive use has been made of the semicolons at the end of each line. The semicolon simply represses output to the computer screen, saving valuable time and clutter. The progression of the root finding algorithm is given in Table 1.2. An accuracy of 10^{-5} is achieved after 14 iterations.

Table 1.2 Convergence of the bisection iteration scheme to an accuracy of 10^{-5} given the end points of $x_l = -4$ and $x_r = -2.8$.

j	xc	fc
1	−3.4000	−0.2977
2	−3.1000	0.0034
3	−2.9500	−0.1416
⋮	⋮	⋮
10	−3.0965	−0.0005
⋮	⋮	⋮
14	−3.0964	0.0000

1.3 Iteration: The Newton–Raphson Method

Iteration methods are of fundamental importance, not only as a mathematical concept, but also as a practical tool for solving many problems that arise in the engineering, physical, and biological sciences. In a large number of applications, the convergence or divergence of an iteration scheme is of critical importance. Iteration schemes also help illustrate the basic functionality of **for** loops and **if** statements. In later chapters, iteration schemes become important for solving many problems which include:

- root finding $f(x) = 0$ (Newton's method);
- linear systems $\mathbf{Ax} = \mathbf{b}$ (system of equations);
- differential equations $d\mathbf{y}/dt = f(\mathbf{y}, t)$.

In the first two applications listed above, the convergence of an iteration scheme is of fundamental importance. For applications in differential equations, the iteration predicts the future state of a given system provided the iteration scheme is stable. The stability of iteration schemes will be discussed in later sections on differential equations.

The generic form of an iteration scheme is as follows:

$$p_{k+1} = g(p_k)\,, \tag{1.3.1}$$

where $g(p_k)$ determines the iteration rule to be applied. Thus, starting with an initial value p_0, we can construct the hierarchy

$$p_1 = g(p_0) \tag{1.3.2a}$$

$$p_2 = g(p_1) \tag{1.3.2b}$$

$$p_3 = g(p_2) \tag{1.3.2c}$$

$$\vdots \tag{1.3.2d}$$

Such an iteration procedure is trivial to implement with a **for** loop structure. Indeed, an example code will be given shortly. Of interest are the values taken by successive iterations p_k. In particular, the values can converge, diverge, become periodic, or vary randomly (chaotically).

1.3.1 Fixed points

Fixed points are fundamental to understanding the behaviour of a given iteration scheme. Fixed points are points defined such that the input is equal to the output in (1.3.1):

$$p_k = g(p_k)\,. \tag{1.3.3}$$

The fixed points determine the overall behaviour of the iteration scheme. Effectively, we can consider two functions

$$y_1 = x \tag{1.3.4a}$$

$$y_2 = g(x) \tag{1.3.4b}$$

and the fixed point occurs for

$$y_1 = y_2 \quad \rightarrow \quad x = g(x)\,. \tag{1.3.5}$$

Defining these quantities in a continuous fashion allows for an insightful geometrical interpretation of the iteration process. Figure 1.2 demonstrates graphically the location of the fixed point for a given $g(x)$. The intersection of $g(x)$ with the line $y = x$ gives the fixed point locations of interest. Note that for a given $g(x)$, multiple fixed points may exist. Likewise, there may be no fixed points for certain iteration functions $g(x)$.

From this graphical interpretation, the concept of convergence or divergence can be easily understood. Figure 1.3 shows the iteration process given by (1.3.1) for four prototypical iteration functions $g(x)$. Illustrated is convergence to and divergence from (both in a

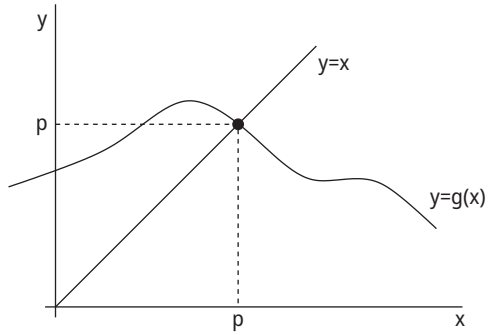


Figure 1.2 Graphical representation of the fixed point. The value p is the point at which $p = g(p)$, i.e. where the input value of the iteration is equal to the output value.

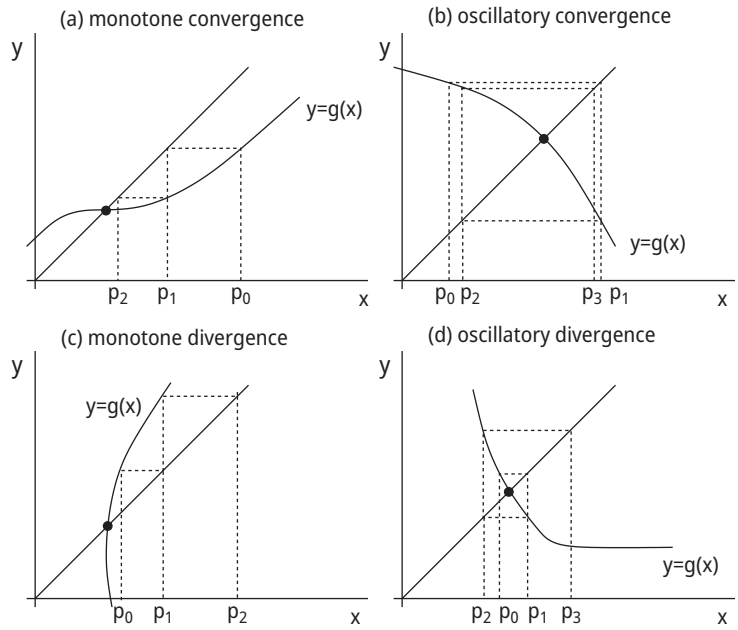


Figure 1.3 Geometric interpretation of the iteration procedure $p_{k+1} = g(p_k)$ for four different functions $g(x)$. Some of the possible iteration behaviours near the fixed point included monotone convergence or divergence and oscillatory convergence or divergence.

monotonic and oscillatory sense) the fixed point, where $x = g(x)$. Note that in practice, when iterating towards a convergent solution, the computations should be stopped once a desired accuracy is achieved. It is not difficult to imagine from Figs. 3(b) and (d) that the iteration may also result in periodicity of the iteration scheme; thus, the ability of iteration to produce periodic orbits and solutions.

1.3.2 The Newton–Raphson method

One of the classic uses of iteration is for finding roots of a given function or polynomial. We therefore develop the basic ideas of the Newton–Raphson iteration method, commonly known as Newton's method. We begin by considering a single nonlinear equation

$$f(x_r) = 0 \tag{1.3.6}$$

where x_r is the root to the equation and the value being sought. We would like to develop a scheme, which systematically determines the value of x_r. The Newton–Raphson method is an iterative scheme, which relies on an initial guess, x_0, for the value of the root. From this guess, subsequent guesses are determined until the scheme either converges to the root x_r or the scheme diverges and another initial guess is used. The sequence of guesses $(x_0, x_1, x_2, \ldots)$ is generated from the slope of the function $f(x)$. The graphical procedure

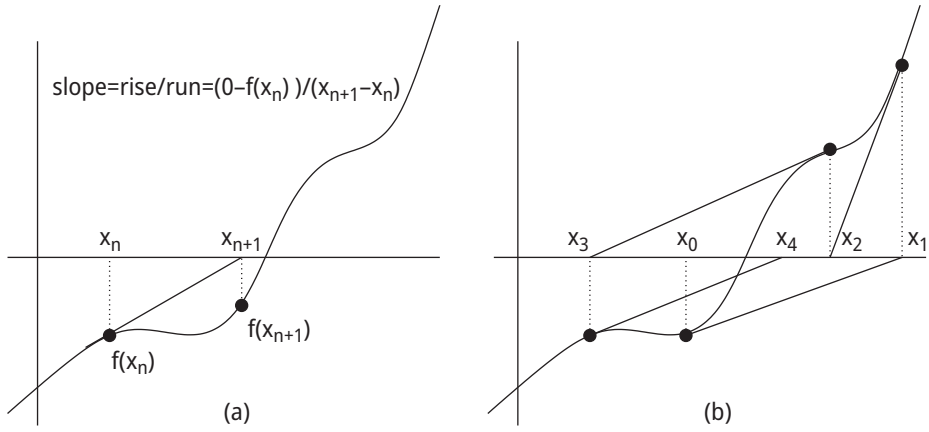


Figure 1.4 Construction and implementation of the Newton–Raphson iteration formula. In (a), the slope is the determining factor in deriving the Newton–Raphson formula. In (b), a graphical representation of the iteration scheme is given.

is illustrated in Fig. 1.4. In essence, everything relies on the slope formula as illustrated in Fig. 1.4(a):

$$\text{slope} = \frac{df(x_n)}{dx} = \frac{\text{rise}}{\text{run}} = \frac{0 - f(x_n)}{x_{n+1} - x_n} \,. \tag{1.3.7}$$

Rearranging this gives the Newton–Raphson iterative relation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \,. \tag{1.3.8}$$

A graphical example of how the iteration procedure works is given in Fig. 1.4(b), where a sequence of iterations is demonstrated. Note that the scheme fails if $f'(x_n) = 0$ since then the slope line never intersects $y = 0$. Further, for certain guesses, the iterations may diverge. Provided the initial guess is sufficiently close, the scheme usually will converge. Conditions for convergence can be found in Burden and Faires [7].

To implement this method, consider once again the example problem of the last section for which we are to determine the roots of $f(x) = \exp(x) - \tan(x) = 0$. In this case, we have

$$f(x) = \exp(x) - \tan(x) \tag{1.3.9a}$$

$$f'(x) = \exp(x) - \sec^2(x) \tag{1.3.9b}$$

so that the Newton iteration formula becomes

$$x_{n+1} = x_n - \frac{\exp(x_n) - \tan(x_n)}{\exp(x_n) - \sec^2(x_n)} \,. \tag{1.3.10}$$

Table 1.3 Convergence of the Newton iteration scheme to an accuracy of 10^{-5} given the initial guess $x_0 = -4$.

j	x(j)	fc
1	−4	1.1761
2	−3.4935	0.3976
3	−3.1335	0.0355
4	−3.0964	2.1946×10^{-6}

This results in the following algorithm for searching for a root of this function.

newton.m

```
x = np.array([-4])  # initial guess
for j in range(1000):
    x = np.append(
        x, x[j]-(np.exp(x[j])-np.tan(x[j]))
        / (np.exp(x[j])-1 / np.cos(x[j]) ** 2))
    fc = np.exp(x[j + 1]) - np.tan(x[j + 1])

    if abs(fc) < 1e-5:
        break

 print(x[j + 1])
 print(fc)
```

Note that the **break** command ejects you from the current loop. In this case, that is the j loop. This effectively stops the iteration procedure from proceeding to the next value of $f(x_n)$. The progression of the root finding algorithm is given in Table 1.3. An accuracy of 10^{-5} is achieved after only four iterations. Compare this to 14 iterations for the bisection method of the last section.

1.4 Function Calls, Input/Output Interactions, and Debugging

A fairly standard way to program in Python is to write most algorithms as functions and to call them through other functions or directly in the Python dialog box. Thus, building functions is one of the most common techniques for code implementation.

In what follows, a number of examples will be used to illustrate how to build functions in Python. The goal is to start with simple functions and to build in sophistication from there. Thus, consider building a function that evaluates the simple function

$$f(x) = x^3 + \sin(x). \tag{1.4.1}$$

Obviously, the function requires a value of x (input) which must be used to evaluate the function. This *input* must be specified in order for the function call to make sense. In Python, this function can be easily implemented by building the following Python script:

```
def myfun(x):
    y = x**3 + np.sin(x)
    return y
```

The dot in the cube of x is for the case that an initial vector is input into the function call. Access to this function comes from a simple command line call such as the following:

```
y = myfun(1);
```

This returns the value of y evaluated at $x = 1$ so that $y = 1^3 + \sin(1) = 1.8415$. Alternatively, one can evaluate the function at multiple values of x by defining the input as a vector as follows:

```
x = np.arange(0,11,1)
y = myfun(x)
```

This yields

```
y =
         0
    1.8415
    8.9093
   27.1411
   63.2432
  124.0411
  215.7206
  343.6570
  512.9894
  729.4121
  999.4560
```

An important task often associated with function calls is the passing of variables or parameters through to the function itself. The following illustrates how to pass a variable along with a parameter. Specifically, assume that we wish to evaluate the function $f(x) = \sin Bx$, where the parameter is a function of the input parameter A so that

$B = A^2 + \cos A$. The following code evaluates the function for a given input vector **x** and parameter A.

```
def funcA(x,A):
    B = A**2 + np.cos(A)
    fA = np.sin(B*x)
    return fA,B
```

Note that `function` returns both the value of the function and the value of the parameter B. Specifically, the code can be executed as

```
fA, B = funcA(np.array([3, 4, 5]), 2)
```

which produces the output

```
fA = -0.9704    0.9804   -0.8018
B = 3.5839
```

Finally, functions can be called within functions, which is often convenient in executing complex codes. The following function **funcC.m** uses two parameters A and B to evaluate a third variable C which is then used to evaluate the final function. Thus, the function has two embedded functions within the called function. There is no limit on how many functions can be embedded within another.

```
def funcC(x)
     A=2; B=3;
     C=f_param(A,B);
     f=func(x,C);
     return f, A, B, C

def f_param(A,B)
     C=A^2+B^2;
     return C

def func(x,C)
     f=C*sin(C*x);
     return f
```

This function can be called by the command line

```
x = np.arange(0,11,1)
f, a, b, c = funcC(x)
```

This returns the vector **f** evaluated at the vector **x** values along with the parameters A, B, and C.

1.4.1 Input and output to Python codes

It is also easy to make Python interactive in nature. Thus, Python can ask for user input as the algorithm progresses through its execution. Consider the function

```
def func(x):
    A = float(input('A= '))
    return np.sin(A * x)
```

This code will pause at the second line and display the following to the screen

```
A=
```

The user then inputs any numerical value of A desired and the function will then continue the function evaluation with this user-specified (and interactive) value of A. Text strings can also be input into the system. The following is an example

```
reply = input('Do you want more? Y/N [Y]: ')
if not reply:
    reply = 'Y'

print(reply)
```

The option `'s'` in the input command allows for the acceptance of a string. The **isempty** line is in case no input is generated and a preassigned value is given to the variable **reply**.

Output to the screen can also be easily generated in a variety of ways. The **disp** (for display) is perhaps the easiest to use. So, for instance, one could use the command

```
display('hello world')
```

to display the test string: *hello world*. Alternatively, one could also simply put a text string in quotes to produce a text string

```
'hello world'
```

Alternatively, one could define a variable that can be viewed at any time by defining, for instance, the following:

```
x = 'hello world'
```

Thus, the variable **x** takes on the value of the text string.

1.5 Plotting and Importing/Exporting Data

The graphical representation of data is a fundamental aspect of any technical scientific communication. Python provides an easy and powerful interface for plotting and representing a large variety of data formats. Like all Python structures, plotting is dependent on the effective manipulation of vectors and matrices.

To begin the consideration of graphical routines, we first construct a set of functions to plot and manipulate. To define a function, the plotting domain must first be specified. This can be done by using a routine which creates a row vector

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-10,10.1,0.1)
y = np.sin(x)
```

Here the row vector $\mathbf{x}$ spans the interval $x \in [-10, 10]$ in steps of $\Delta x = 0.1$. The second command creates a row vector $\mathbf{y}$, which gives the values of the sine function evaluated at the corresponding values of $\mathbf{x}$. A basic plot of this function can then be generated by the command

```
plt.plot(x,y)
```

Note that this graph lacks a title or axis labels. These are important in generating high-quality graphics.

The preceding example considers only equally spaced points. However, Python will generate function values for any specified coordinate values. For instance, the two lines

```
x2 = [-5,np.sqrt(3),np.pi,4]
y2 = np.sin(x2)
```

will generate the values of the sine function at $x = -5, -\sqrt{3}$, and π. The **linspace** command is also helpful in generating a domain for defining and evaluating functions. The basic procedure for this function is as follows:

```
x3 = np.linspace(-10,10,64)
y3 = x3*np.sin(x3)
```

This will generate a row vector $\mathbf{x}$ which goes from -10 to 10 in 64 steps. This can often be a helpful function when considering intervals, where the number of discretized steps gives a complicated Δx. In this example, we are considering the function $x \sin x$ over the interval $x \in [-10, 10]$. By doing so, the period must be included before the multiplication sign. This will then perform a *component-by-component* multiplication, thus creating a row vector with the values of $x \sin x$.

To plot all of the above data sets in a single plot, we can do either one of the following routines.

```
plt.plot(x,y)
plt.plot(x2,y2)
plt.plot(x3,y3)
```

In this case, the **hold on** command is necessary after the first plot so that the second plot command will not overwrite the first data plotted. This will plot all three data sets on the same graph with a default blue line. This graph will be *figure 1*. Any subsequent plot commands will plot the new data on top of the current data until the **hold off** command is executed. Alternatively, all the data sets can be plotted on the same graph with

```
plt.plot(x,y,x2,y2,x3,y3)
```

For this case, the three pairs of vectors are prescribed within a single plot command. This figure generated will be *figure 2*. An advantage to this method is that the three data sets will be of different colours, which is better than having them all in the default colour of blue.

This is only the beginning of the plotting and visualization process. Many aspects of the graph must be augmented with speciality commands in order to more accurately relay the information. Of significance is the ability to change the line colours and styles of the plotted data. By using the **help plot** command, a list of options for customizing data representation is given. In the following, a new figure is created which is customized as follows

```
plt.plot(x,y,x2,y2,'g*',x3,y3,'mo:')
```

Table 1.4 Plotting options which can be used in a standard **plot** command. The format would typically be, for instance, **plot(x,y,'r:h')** which would put a red dotted line through the data points marked with a hexagram.

line style	colour	symbol
- = solid	k = black	. = point
: = dotted	b = blue	o = circle
-. = dashed-dot	r = red	x = x-mark
- - = dashed	c = cyan	+ = plus
(none) = no line	m = magenta	* = star
	y = yellow	s = square
	g = green	D = diamond
	w = white	v = triangle (down)
		∧ = triangle (up)
		< = triangle (left)
		> triangle (right)
		p = pentagram
		h = hexagram

This will create the same plot as in *figure 2*, but now the second data set is represented by green stars and the third data set is represented by a magenta dotted line with the actual data points given by a magenta hollow circle. This kind of customization greatly helps distinguish the various data sets and their individual behaviours. An example of this plot is given in Fig. 1.5. Extra features, such as axis label and titles, are discussed in the following paragraphs. Table 1.4 gives a list of options available for plotting different line styles, colours, and symbol markers.

Labelling the axis and placing a title on the figure is also of fundamental importance. This can be easily accomplished with the commands

```
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('Example Graph')
```

The strings given within the ' sign are now printed in a centred location along the x-axis, y-axis, and title location, respectively.

A legend is then important to distinguish and label the various data sets, which are plotted together on the graph. Within the **legend** command, the position of the legend can be easily specified. The command **help legend** gives a summary of the possible legend placement positions, one of which is to the right of the graph and does not interfere with the graph data. To place a legend in the above graph in an *optimal* location, the following command is used.

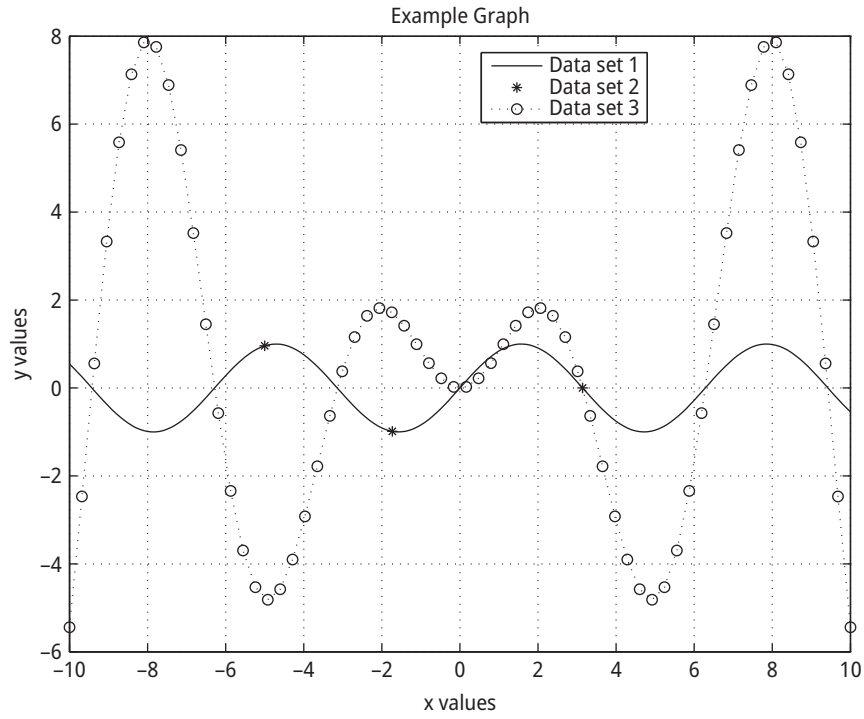


Figure 1.5 Plot of the functions $f(x) = \sin(x)$ and $f(x) = x\sin(x)$. The stars ($*$) are the value of $\sin(x)$ at the locations $x = \{-5, \sqrt{3}, \pi, 4\}$. The **grid on** and **legend** command have both been used. Note that the default font size is smaller than would be desired. More advanced plot settings and adjustments are considered in Chapter 7 on visualization.

```
plt.plot(x,y,label='Data Set 1')
plt.plot(x2,y2,label='Data Set 2')
plt.plot(x3,y3,label='Data Set 3')
plt.legend()
```

Here, the strings correspond to the three plotted data sets in the order they were plotted. The location command at the end of the **legend** command is the option setting for placement of the legend box. In this case, the option *Best* tries to pick the best possible location, which does not interfere with any of the data. Table 1.5 gives a list of the location options that can be used in Python. And in addition to these specified locations, a 1×4 vector may be placed in this slot with the specific location desired for the legend.

Table 1.5 Legend position placement options. The default is to place the legend on the inside top right of the graph. One can also place a 1×4 vector in the specification to manually give a placement of the legend.

Python position specification	legend position
'North'	inside plot box near top
'South'	inside bottom
'East'	inside right
'West'	inside left
'NorthEast'	inside top right (default)
'NorthWest'	inside top left
'SouthEast'	inside bottom right
'SouthWest'	inside bottom left
'NorthOutside'	outside plot box near top
'SouthOutside'	outside bottom
'EastOutside'	outside right
'WestOutside'	outside left
'NorthEastOutside'	outside top right
'NorthWestOutside'	outside top left
'SouthEastOutside'	outside bottom right
'SouthWestOutside'	outside bottom left
'Best'	least conflict with data in plot
'BestOutside'	least unused space outside plot

1.5.1 Subplots

Another possible method for representing data is with the **subplot** command. This allows for multiple graphs to be arranged in a row and column format. In this example, we have three data sets which are under consideration. To plot each data set in an individual subplot requires three plot frames. Thus, we construct a plot containing three rows and a single column. The format for this command structure is as follows:

```
fig = plt.subplot(311)
plt.plot(x,y), plt.axis([-10, 10, -10, 10])

fig = plt.subplot(312)
plt.plot(x2,y2,'g*'), plt.axis([-10, 10, -10, 10])

fig = plt.subplot(313)
plt.plot(x3,y3,'mo:')
```

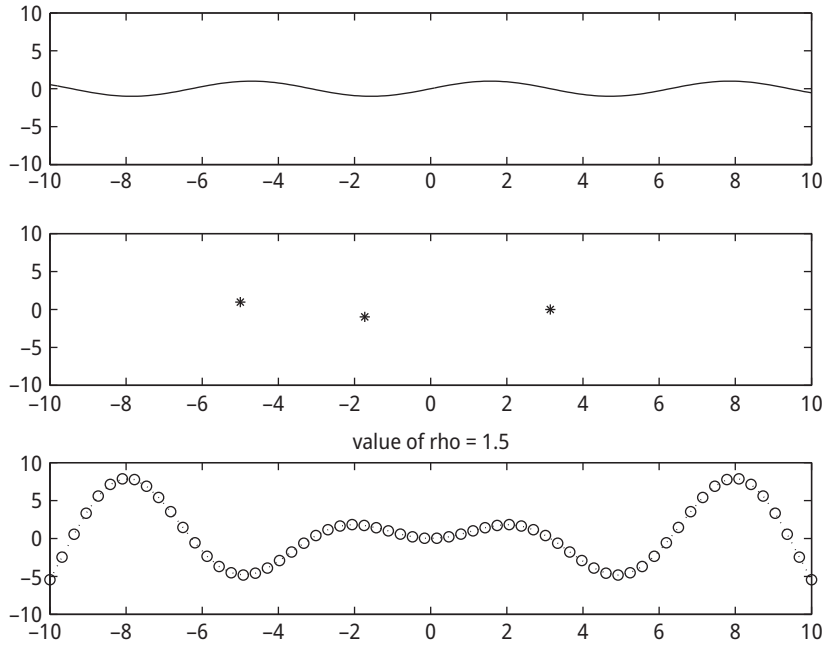


Figure 1.6 Subplots of the functions $f(x) = \sin(x)$ and $f(x) = x\sin(x)$. The stars (*) in the middle plot are the values of $\sin(x)$ at the locations $x = \{-5, \sqrt{3}, \pi, 4\}$. The **str** command has been used for the title of the bottom plot.

Note that the **subplot** command is of the form **subplot(row, column, graph number)**, where the graph number refers to the current graph being considered. In this example, the axis command has also been used to make all the graphs have the same x and y ranges. The command structure for the axis command is **axis([xmin xmax ymin ymax])**. For each subplot, the use of the **legend**, **xlabel**, **ylabel**, and **title** command can be used. An example of this plot is given in Fig. 1.6.

Remark 1: All the graph customization techniques discussed can be performed directly within the Python graphics window. Simply go to the edit button and choose to edit axis properties. This will give full control of all the axis properties discussed so far and more. However, once Python is shut down or the graph is closed, all the customization properties are lost, and you must start again from scratch. This gives an advantage to developing a nice set of commands in a .m file to customize your graphics.

Remark 2: To put a grid on the graph, simply use the **grid on** command. To take it off, use **grid off**. The number of grid lines can be controlled from the editing options in the graphics window. This feature can aid in illustrating more clearly certain phenomena and should be considered when making plots.

Remark 3: To put a variable value in a string, the **str** (number to string) command can be used. For instance, the code

```
a=5
plt.title('value of a=' + str(a))
```

creates a title which reads "value of rho = 1.5". Thus, this command converts variable values into useful string configurations.

Remark 4: More advanced plot settings, adjustments, and modifications are considered in Chapter 7.

1.5.2 Load, save, and print

Once a graph has been made or data generated, it may be useful to save the data or print the graphs for inclusion in a write-up or presentation. The **save** and **print** commands are the appropriate commands for performing such actions.

The **save** command will write any data out to a file for later use in a separate program or for later use in Python. To save workspace variables to a file, the following command structure is used

```
np.save('file.npy', array_to_save)
```

This will save an array variable in your workspace to an npy array named **file.npy**. This is an extremely powerful and easy command to use. However, you can only access the data again through Python. To recall this data at a future time, simply load it back into Python with the command

```
X = np.load('file.npy')
```

This will reload into the workspace the saved array variable saved in **file.npy**. This command is ideal when closing down operations on Python and resuming at a future time. Alternatively, it may be advantageous to save data for use in a different software package. There are a number of architectures to do this, and it is best to consult stackoverflow in order to learn how to do this for a specific file format. To save multiple files in Python, a **pickle** file can be created. Such a file saves data in a data dictionary. The following is an example of how to construct a **.pkl** file that contains the three numpy arrays **X1**, **X2**, and **X3**

```
import pickle
data_dict = {'array1': X1, 'array2': X2, 'array3': X3}

# Serialize the dictionary to a .pkl file
with open('data.pkl', 'wb') as f:
    pickle.dump(data_dict, f)

# Deserialize the dictionary from the .pkl file
with open('data.pkl', 'rb') as f:
    loaded_data_dict = pickle.load(f)

# Access loaded arrays
Y1 = loaded_data_dict['array1']
Y2 = loaded_data_dict['array2']
Y3 = loaded_data_dict['array3']
```

This saves and then reloads the **data.pkl** file that contains the three numpy arrays as **Y1**, **Y2**, and **Y3**. This saving option is advantageous when considering the packaging up of a number of data files for later use. The data can take any form, not just numpy arrays, i.e. you can package arrays of text strings.

It is also necessary at times to save file names according to a loop variable. For instance, you may create a loop variable, which executes a Python script a number of times with a different key parameter, which is controlled by the loop variable. In this case, it is often imperative to save the variables according to the loop name, as repeated passage through the loop will overwrite the variables you wish to save. The following example illustrates the saving procedure:

```
for loop in range(1, 6):
    (expression to execute)
    filename = f"loopnumber{loop}.npy"
    np.save(filename, array_to_save)
```

Subsequent passes through the loop will save files called **loopnumber1.npy**, **loopnumber2.npy**, etc. This can often be helpful for running scripts which generate the same common variable names in a loop.

If you desire to print a figure to a file, the **savefig** command needs to be utilized. There a large number of graphics formats which can be used. A common graphics format would involve the command

```
plt.savefig('fig.pdf', format='pdf')
```

Table 1.6 Some figure export options for Python. Most common output options are available which can then be imported to word, PowerPoint, or latex.

figure output format	print format option
PDF	pdf
encapsulated postscript	eps
JPEG image	jpg
TIFF image	tiff
portable network graphics	png

which will print the current figure as a pdf file named **fig.pdf**. A list of some of the common figure export options can be found in Table 1.6. As with saving according to a loop variable name, batch printing can also be easily performed. The command structure

```
for j in range(5):
   (plot expressions)
   plt.savefig(f'figure{j}.png', dpi=80)
```

prints the plots generated in the j loop as **figure1.png**, **figure2.png**, etc. This is a very useful command for generating batch printing.

2 Linear Systems

2.1 Direct Solution Methods for $\mathbf{Ax} = \mathbf{b}$ 38
2.1.1 LU decomposition 40
2.1.2 The permutation matrix 42
2.1.3 Python commands 42
2.2 Iterative Solution Methods for $\mathbf{Ax} = \mathbf{b}$ 43
2.2.1 Strictly diagonal dominant 44
2.2.2 Modification and enhancements: Gauss–Seidel 45
2.3 Gradient (Steepest) Descent for $\mathbf{Ax} = \mathbf{b}$ 46
2.4 Eigenvalues, Eigenvectors, and Solvability 51
2.4.1 Matrix powers 55
2.4.2 Solvability and the Fredholm-alternative theorem 56
2.5 Eigenvalues and Eigenvectors for Face Recognition 56
2.6 Nonlinear Systems 64

The solution of linear systems is one of the most basic aspects of computational science. In many applications, the solution technique often gives rise to a system of linear equations, which need to be solved as efficiently as possible. In addition to Gaussian elimination, there are a host of other techniques, which can be used to solve a given problem. This chapter offers an overview of these methods and techniques. Ultimately, the goal is to solve large linear systems of equations as quickly as possible. Thus, consideration of the operation count is critical.

2.1 Direct Solution Methods for $\mathbf{Ax} = \mathbf{b}$

A central concern in almost any computational strategy is a fast and efficient computational method for achieving a solution of a large system of equations $\mathbf{Ax} = \mathbf{b}$. In trying to render a computation tractable, it is crucial to minimize the operations it takes in solving such a system. There are a variety of direct methods for solving $\mathbf{Ax} = \mathbf{b}$: Gaussian elimination, LU decomposition, and inverting the matrix $\mathbf{A}$. In addition to these direct methods, iterative

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0003

schemes can also provide efficient solution techniques. Some basic iterative schemes will be discussed in what follows.

The standard beginning to discussions of solution techniques for $\mathbf{Ax} = \mathbf{b}$ involves Gaussian elimination. We will consider a very simple example of a 3×3 system in order to understand the operation count and numerical procedure involved in this technique. As a specific example, consider the three equations and three unknowns

$$x_1 + x_2 + x_3 = 1 \tag{2.1.1a}$$

$$x_1 + 2x_2 + 4x_3 = -1 \tag{2.1.1b}$$

$$x_1 + 3x_2 + 9x_3 = 1. \tag{2.1.1c}$$

In matrix algebra form, we can rewrite this as $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}. \tag{2.1.2}$$

The Gaussian elimination procedure begins with the construction of the augmented matrix

$$\begin{aligned} [\mathbf{A}|\mathbf{b}] &= \left[\begin{array}{ccc|c} \underline{\mathbf{1}} & 1 & 1 & 1 \\ 1 & 2 & 4 & -1 \\ 1 & 3 & 9 & 1 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 2 & 8 & 0 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & \underline{\mathbf{1}} & 3 & -2 \\ 0 & 1 & 4 & 0 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 1 & 2 \end{array}\right] \end{aligned} \tag{2.1.3}$$

where we have underlined and bolded the pivot of the augmented matrix. Back-substituting then gives the solution

$$x_3 = 2 \qquad \rightarrow \; x_3 = 2 \tag{2.1.4a}$$

$$x_2 + 3x_3 = -2 \qquad \rightarrow \; x_2 = -8 \tag{2.1.4b}$$

$$x_1 + x_2 + x_3 = 1 \qquad \rightarrow \; x_1 = 7\,. \tag{2.1.4c}$$

This procedure can be carried out for any matrix $\mathbf{A}$ which is nonsingular, i.e. $\det \mathbf{A} \neq 0$. In this algorithm, we simply need to avoid these singular matrices and occasionally shift the rows to avoid a zero pivot. Provided we do this, it will always yield a unique answer.

In scientific computing, the fact that this algorithm works is secondary to the concern of the time required in generating a solution. The operation count for the Gaussian elimination can easily be estimated from the algorithmic procedure for an $N \times N$ matrix:

(1) Movement down the N pivots.

(2) For each pivot, perform N additions/subtractions across the columns.

(3) For each pivot, perform the addition/subtraction down the N rows.

In total, this results in a scheme whose operation count is $O(N^3)$. The back-substitution algorithm can similarly be calculated to give an $O(N^2)$ scheme.

2.1.1 LU decomposition

Each Gaussian elimination operation costs $O(N^3)$ operations. This can be computationally prohibitive for large matrices when repeated solutions of $\mathbf{Ax} = \mathbf{b}$ must be found. When working with the same matrix $\mathbf{A}$, however, the operation count can easily be brought down to $O(N^2)$ using LU factorization which splits the matrix $\mathbf{A}$ into a lower triangular matrix L, and an upper triangular matrix U. For a 3×3 matrix, the LU factorization scheme splits $\mathbf{A}$ as follows:

$$\mathbf{A} = \mathbf{LU} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}. \tag{2.1.5}$$

Thus, the $\mathbf{L}$ matrix is lower triangular and the $\mathbf{U}$ matrix is upper triangular. This then gives

$$\mathbf{Ax} = \mathbf{b} \;\rightarrow\; \mathbf{LUx} = \mathbf{b} \tag{2.1.6}$$

where by letting $\mathbf{y} = \mathbf{Ux}$, we find the coupled system

$$\mathbf{Ly} = \mathbf{b} \quad \text{and} \quad \mathbf{Ux} = \mathbf{y}. \tag{2.1.7}$$

The system $\mathbf{Ly} = \mathbf{b}$

$$y_1 = b_1 \tag{2.1.8a}$$

$$m_{21} y_1 + y_2 = b_2 \tag{2.1.8b}$$

$$m_{31} y_1 + m_{32} y_2 + y_3 = b_3 \tag{2.1.8c}$$

can be solved by $O(N^2)$ forward-substitution and the system $\mathbf{Ux} = \mathbf{y}$

$$u_{11}x_1 + u_{12}x_2 + u_{13}x_3 = y_1 \tag{2.1.9a}$$

$$u_{22}x_2 + u_{23}x_3 = y_2 \tag{2.1.9b}$$

$$u_{33}x_3 = y_3 \tag{2.1.9c}$$

can be solved by $O(N^2)$ back-substitution. Thus, once the factorization is accomplished, the LU results in an $O(N^2)$ scheme for arriving at the solution. The factorization itself is $O(N^3)$, but you only have to do this once. Note, you should **always** use LU decomposition if possible. Otherwise, you are doing far more work than necessary in achieving a solution.

As an example of the application of the LU factorization algorithm, we consider the 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.1.10}$$

The factorization starts from the matrix multiplication of the matrix $\mathbf{A}$ and the identity matrix $\mathbf{I}$

$$\mathbf{A} = \mathbf{IA} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \underline{\mathbf{4}} & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{pmatrix}. \tag{2.1.11}$$

The factorization begins with the pivot element. To use Gaussian elimination, we would multiply the pivot by $-1/2$ to eliminate the first column element in the second row. Similarly, we would multiply the pivot by $1/4$ to eliminate the first column element in the third row. These multiplicative factors are now part of the first matrix above:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & \underline{\mathbf{-2.5}} & 4.5 \\ 0 & 1.25 & 6.25 \end{pmatrix}. \tag{2.1.12}$$

To eliminate on the third row, we use the next pivot. This requires that we multiply by $-1/2$ in order to eliminate the second column, third row. Thus, we find

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.1.13}$$

Thus, we find that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U} = \begin{pmatrix} 4 & 3 & -1 \\ 0 & -2.5 & 4.5 \\ 0 & 0 & 8.5 \end{pmatrix}. \tag{2.1.14}$$

It is easy to verify by direct multiplication that indeed $\mathbf{A} = \mathbf{LU}$. Just like Gaussian elimination, the cost of factorization is $O(N^3)$. However, once $\mathbf{L}$ and $\mathbf{U}$ are known, finding the solution is an $O(N^2)$ operation.

2.1.2 The permutation matrix

As will often happen with Gaussian elimination, following the above algorithm will at times result in a zero pivot. This is easily handled in Gaussian elimination by shifting rows in order to find a nonzero pivot. However, in LU decomposition, we must keep track of this row shift since it will effect the right-hand side vector $\mathbf{b}$. We can keep track of row shifts with a row permutation matrix $\mathbf{P}$. Thus, if we need to permute two rows, we find

$$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{PAx} = \mathbf{Pb} \rightarrow \mathbf{LUx} = \mathbf{Pb} \tag{2.1.15}$$

thus $\mathbf{PA} = \mathbf{LU}$ where $\mathbf{PA}$ no longer has any zero pivots. To shift rows one and two, for instance, we would have

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & & & \end{pmatrix}. \tag{2.1.16}$$

Thus, the permutation matrix starts with the identity matrix. If we need to shift rows j and k, then we shift these corresponding rows in the permutation matrix $\mathbf{P}$. If permutation is necessary, Python can supply the permutation matrix associated with the LU decomposition.

2.1.3 Python commands

The commands for executing the linear system solve are as follows:

```
import numpy as np
import scipy
import scipy.linalg

A = np.array([[1,1,1],[1,2,4],[1,3,9]])
b = np.array([[1],[-1],[1]])

x = np.linalg.solve(A,b)

P, L, U = scipy.linalg.lu(A)

b2 = np.matmul(P, b)
y = np.linalg.solve(L,b2)
x = np.linalg.solve(U,y)
```

2.2 Iterative Solution Methods for $\mathbf{Ax} = \mathbf{b}$

In addition to the standard techniques of Gaussian elimination or LU decomposition for solving $\mathbf{Ax} = \mathbf{b}$, a wide range of iterative techniques is available. These iterative techniques can often go under the name of Krylov space methods [8]. The idea is to start with an initial guess for the solution and develop an iterative procedure that will converge to the solution. The simplest example of this method is known as a Jacobi iteration scheme. The implementation of this scheme is best illustrated with an example. We consider the linear system

$$4x - y + z = 7 \tag{2.2.1a}$$

$$4x - 8y + z = -21 \tag{2.2.1b}$$

$$-2x + y + 5z = 15. \tag{2.2.1c}$$

We can rewrite each equation as follows

$$x = \frac{7 + y - z}{4} \tag{2.2.2a}$$

$$y = \frac{21 + 4x + z}{8} \tag{2.2.2b}$$

$$z = \frac{15 + 2x - y}{5}. \tag{2.2.2c}$$

To solve the system iteratively, we can define the following Jacobi iteration scheme based on the above

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.2.3a}$$

$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.2.3b}$$

$$z_{k+1} = \frac{15 + 2x_k - y_k}{5}. \tag{2.2.3c}$$

An algorithm is then easily implemented computationally. In particular, we would follow the structure:

(1) Guess initial values: (x_0, y_0, z_0).

(2) Iterate the Jacobi scheme: $\mathbf{x}_{k+1} = \mathbf{D}^{-1}((\mathbf{D} - \mathbf{A})\mathbf{x}_k + \mathbf{b})$.

(3) Check for convergence: $\| \mathbf{x}_{k+1} - \mathbf{x}_k \|$<tolerance.

Note that the choice of an initial guess is often critical in determining the convergence to the solution. Thus, the more that is known about what the solution is supposed to look like, the higher the chance of successful implementation of the iterative scheme. Although

Table 2.1 Convergence of Jacobi iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.375	3.0
2	1.84375	3.875	3.025
⋮	⋮	⋮	⋮
15	1.99999993	3.99999985	2.9999993
⋮	⋮	⋮	⋮
19	2.0	4.0	3.0

there is no reason *a priori* to believe this iteration scheme would converge to the solution of the corresponding system $\mathbf{Ax} = \mathbf{b}$, Table 2.1 shows that indeed this scheme converges remarkably quickly to the solution for this simple example.

Given the success of this example, it is easy to conjecture that such a scheme will always be effective. However, we can reconsider the original system by interchanging the first and last set of equations. This gives the system

$$-2x + y + 5z = 15 \tag{2.2.4a}$$

$$4x - 8y + z = -21 \tag{2.2.4b}$$

$$4x - y + z = 7. \tag{2.2.4c}$$

To solve the system iteratively, we can define the following Jacobi iteration scheme based on this rearranged set of equations

$$x_{k+1} = \frac{y_k + 5z_k - 15}{2} \tag{2.2.5a}$$

$$y_{k+1} = \frac{21 + 4x_k + z_k}{8} \tag{2.2.5b}$$

$$z_{k+1} = y_k - 4x_k + 7. \tag{2.2.5c}$$

Of course, the solution should be exactly as before. However, Table 2.2 shows that applying the iteration scheme leads to a set of values which grow to infinity. Thus, the iteration scheme quickly fails.

2.2.1 Strictly diagonal dominant

The difference in the two Jacobi schemes above involves the iteration procedure being strictly diagonal dominant. We begin with the definition of strict diagonal dominance. A matrix $\mathbf{A}$ is strictly diagonal dominant if for each row, the sum of the absolute values of the off-diagonal terms is less than the absolute value of the diagonal term:

Table 2.2 Divergence of Jacobi iteration scheme from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	−1.5	3.375	5.0
2	6.6875	2.5	16.375
3	34.6875	8.015625	−17.25
⋮	⋮	⋮	⋮
	$\pm\infty$	$\pm\infty$	$\pm\infty$

$$|a_{kk}| > \sum_{j=1, j\neq k}^{N} |a_{kj}|. \tag{2.2.6}$$

Strict diagonal dominance has the following consequence; given a strictly diagonal dominant matrix $\mathbf{A}$, then an iterative scheme for $\mathbf{Ax} = \mathbf{b}$ converges to a unique solution $\mathbf{x} = \mathbf{p}$. Jacobi iteration produces a sequence $\mathbf{p}_k$ that will converge to $\mathbf{p}$ for any $\mathbf{p}_0$. For the two examples considered here, this property is crucial. For the first example (2.2.1), we have

$$\mathbf{A} = \begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |4| > |-1| + |1| = 2 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |5| > |2| + |1| = 3, \end{array} \tag{2.2.7}$$

which shows the system to be strictly diagonal dominant and guaranteed to converge. In contrast, the second system (2.2.4) is not strictly diagonal dominant as can be seen from

$$\mathbf{A} = \begin{pmatrix} -2 & 1 & 5 \\ 4 & -8 & 1 \\ 4 & -1 & 1 \end{pmatrix} \rightarrow \begin{array}{l} \text{row 1: } |-2| < |1| + |5| = 6 \\ \text{row 2: } |-8| > |4| + |1| = 5 \\ \text{row 3: } |1| < |4| + |-1| = 5. \end{array} \tag{2.2.8}$$

Thus, this scheme is not guaranteed to converge. Indeed, it diverges to infinity.

2.2.2 Modification and enhancements: Gauss–Seidel

It is sometimes possible to enhance the convergence of a scheme by applying modifications to the basic Jacobi scheme. For instance, the Jacobi scheme given by (2.2.3) can be enhanced by the following modifications

$$x_{k+1} = \frac{7 + y_k - z_k}{4} \tag{2.2.9a}$$

$$y_{k+1} = \frac{21 + 4x_{k+1} + z_k}{8} \tag{2.2.9b}$$

$$z_{k+1} = \frac{15 + 2x_{k+1} - y_{k+1}}{5}. \tag{2.2.9c}$$

Table 2.3 Convergence of Gauss–Seidel iteration scheme to the solution value of $(x, y, z) = (2, 4, 3)$ from the initial guess $(x_0, y_0, z_0) = (1, 2, 2)$.

k	x_k	y_k	z_k
0	1.0	2.0	2.0
1	1.75	3.75	2.95
2	1.95	3.96875	2.98625
⋮	⋮	⋮	⋮
10	2.0	4.0	3.0

Here use is made of the supposedly improved value x_{k+1} in the second equation and x_{k+1} and y_{k+1} in the third equation. This is known as the Gauss–Seidel scheme. Table 2.3 shows that the Gauss–Seidel procedure converges to the solution in half the number of iterations used by the Jacobi scheme in later sections.

Like the Jacobi scheme, the Gauss–Seidel method is guaranteed to converge in the case of strict diagonal dominance. Further, the Gauss–Seidel modification is only one of a large number of possible changes to the iteration scheme which can be implemented in an effort to enhance convergence. It is also possible to use several previous iterations to achieve convergence. Krylov space methods [8] are often high-end iterative techniques especially developed for rapid convergence. Included in these iteration schemes are conjugate gradient methods and generalized minimum residual methods which we will discuss and implement [8].

2.3 Gradient (Steepest) Descent for $\mathbf{Ax} = \mathbf{b}$

The Jacobi and Gauss–Seidel iteration schemes are only two potential iteration schemes that can be developed for solving $\mathbf{Ax} = \mathbf{b}$. But when it comes to high-performance computing and solving extremely large systems of equations, more efficient algorithms are needed to overcome the computational costs of the methods discussed so far. In fact, modern iterative techniques are perhaps the most critical algorithms developed to date for solving large systems which would otherwise be computationally intractable. In this section, the gradient descent, or steepest descent, algorithm is developed. It illustrates how one might engineer an algorithm that continually looks to iterate, in an efficient and intuitive manner, towards the solution of the linear problem $\mathbf{Ax} = \mathbf{b}$. Moreover, it hints at the kind of underlying algorithms and techniques that dominate many high-performance computing applications today including the bi-conjugate gradient descent method (**bicgstab**) and generalized method of residuals (**gmres**) mentioned in the last section.

To begin, we will consider the concept of the *quadratic form*

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x} + \mathbf{c} \tag{2.3.1}$$

where $\mathbf{A}$ is a matrix, $\mathbf{b}$ and $\mathbf{x}$ are vectors, and c is a scalar constant. Note that the quadratic form produces a scalar value $f(\mathbf{x})$.

The gradient of the quadratic form is defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{bmatrix}. \tag{2.3.2}$$

The gradient is a standard quantity from vector calculus that produces a vector field that points in the direction of greatest increase of the function $f(\mathbf{x})$.

The gradient of the function (2.3.1) can be easily computed to be

$$\nabla f(\mathbf{x}) = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b}. \tag{2.3.3}$$

If the matrix $\mathbf{A}$ is symmetric, then $\mathbf{A}^T = \mathbf{A}$ and the gradient has a critical point (maximum or minimum) when $\nabla f = 0$ and $\mathbf{A}\mathbf{x} = \mathbf{b}$:

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = 0. \tag{2.3.4}$$

Thus, the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ is a critical point of the gradient. If, in addition, the matrix $\mathbf{A}$ is *positive definite* ($\mathbf{A}^T\mathbf{x} > 0$), then the solution is a *minimum* of $f(\mathbf{x})$. These observations concerning positive definite and symmetric matrices are critical in the development of the gradient descent technique, i.e. a technique that will try to take advantage of knowing the greatest direction of increase (decrease) of the quadratic form. If $\mathbf{A}$ is not symmetric, then the gradient descent algorithm finds the solution to the linear system $(1/2)(\mathbf{A}^T + \mathbf{A})\mathbf{x} = \mathbf{b}$.

To develop an intuitive feel for the gradient descent technique and the importance of positive definite matrices, consider the following illustrative example system that is both positive definite and symmetric [9]

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \quad c = 0. \tag{2.3.5}$$

With this $\mathbf{A}$ and $\mathbf{b}$, solving for $\mathbf{x}$ is a fairly trivial exercise which yields the solution $\mathbf{x} = [x_1 \;\; x_2]^T = [2 \;\; -2]^T$. Our objective is to develop an iterative scheme, which moves towards this solution in an efficient manner given an initial guess. First, we calculate the quadratic form for this case to be

$$f(\mathbf{x}) = \frac{1}{2}[x_1 \;\; x_2] \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - [2 \;\; -8] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{3}{2}x_1^2 + 2x_1x_2 + 3x_2^2 - 2x_1 + 8x_2 \tag{2.3.6}$$

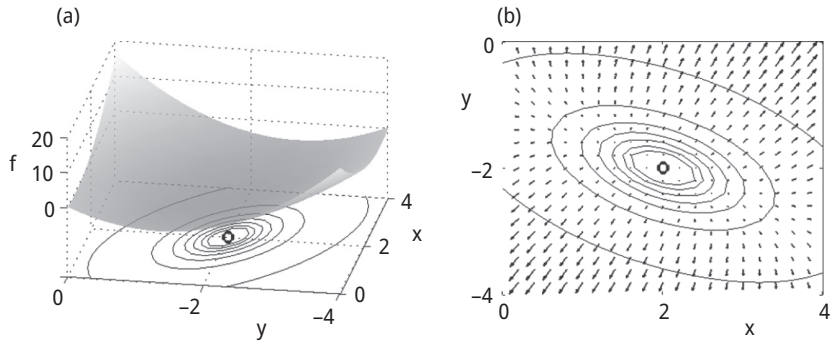


Figure 2.1 Graphical depiction of the gradient descent algorithm. The quadratic form surface $f = (3/2)x_1^2 + 2x_1x_2 + 3x_2^2 - 2x_1 + 8x_2$ is plotted along with its contour lines. In (a), the surface is plotted along with the contour plot beneath. The objective of gradient descent is to find the bottom of the paraboloid. In (b), the gradient is calculated and shown with a quiver plot.

with the gradient

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 3x_1 + 2x_2 - 2 \\ 2x_1 + 6x_2 + 8 \end{bmatrix}. \tag{2.3.7}$$

Figure 2.1 shows the quadratic form surface $f(\mathbf{x})$ along with its contour plot and its gradient. The solution to $\mathbf{Ax} = \mathbf{b}$ is shown as the circle at $\mathbf{x} = [2 \quad -2]^T$. As a result of the matrix being positive definite, the surface is seen to be a paraboloid, whose minimum is at the desired solution. Thus, the idea will be used to construct an algorithm that marches to the bottom of the paraboloid in an efficient manner. If the matrix is not positive definite, then the gradient algorithm is unlikely to work. Figure 2.2 shows four examples that are important to consider: a positive definite matrix, a negative definite matrix, an indefinite (saddle) matrix, and a singular matrix. For the negative definite matrix, the gradient descent algorithm would *fall away* from the maximum point. Similarly for the saddle or singular matrix, the algorithm would either converge to a line of solutions (singular case), or drop away along the saddle.

Given this graphical interpretation of the gradient descent algorithm, one can see how it may generalize to higher dimensions and larger systems. In particular, for a high-dimensional system, one would construct a hyper-paraboloid for a positive definite matrix, where the algorithm would force, in an optimal way, the solution to the bottom of the paraboloid and the solution. The positive definite restriction is similar to the Jacobi algorithm only being guaranteed to work for strictly diagonal dominant matrices.

To build an algorithm that takes advantage of the gradient, an initial guess point is first given and the gradient $\nabla f(\mathbf{x})$ computed. This gives the steepest descent towards the minimum point of $f(\mathbf{x})$, i.e. the minimum is located in the direction given by $-\nabla f(\mathbf{x})$. Note that the gradient does not point at the minimum, but rather gives the steepest path for minimizing $f(\mathbf{x})$. The geometry of the steepest descent suggests the construction of an algorithm whereby the next point of iteration is picked by following the steepest descent so that

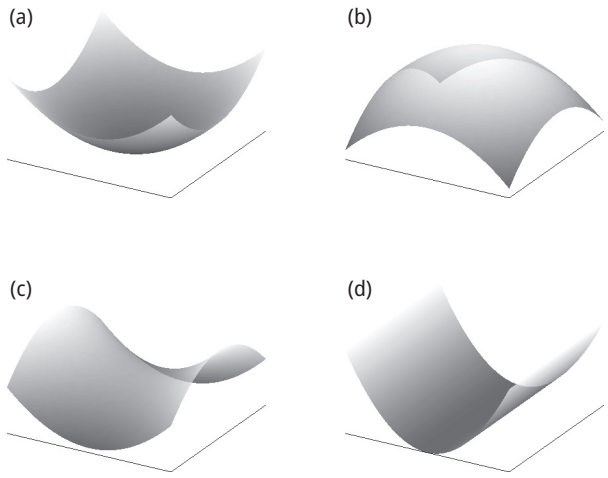


Figure 2.2 Four characteristic matrix forms: (a) positive definite, (b) negative definite, (c) saddle, and (d) singular matrix. Intuition would suggest that a descent algorithm will only work for a positive definite matrix.

$$\xi(\tau) = \mathbf{x} - \tau \nabla f(\mathbf{x}) \tag{2.3.8}$$

where the parameter τ dictates how far to move along the gradient descent curve. In gradient descent, it is crucial to determine when this bottom is reached so that the algorithm is always *going downhill* in an optimal way. This requires the determination of the correct value of τ in the algorithm.

To compute the value of τ, consider the construction of a new function

$$F(\tau) = f(\xi(\tau)) \tag{2.3.9}$$

which must be minimized now as a function of τ. This is accomplished by computing $dF/d\tau = 0$. Thus, one finds

$$\frac{\partial F}{\partial \tau} = -\nabla f(\xi) \nabla f(\mathbf{x}) = 0\,. \tag{2.3.10}$$

The geometrical interpretation of this result is the following: $\nabla f(\mathbf{x})$ is the gradient direction of the current iteration point and $\nabla f(\xi)$ is the gradient direction of the future point, thus τ is chosen so that the two gradient directions are orthogonal.

The following Python code performs a gradient descent search on the linear problem given by (2.3.5). Note that we have already calculated the quadratic form to be minimized along with its gradient.

```
def tausearch(tau):
    x0 = x - tau*(3*x + 2*y - 2)
    y0 = y - tau*(2*x + 6*y + 8)
    mintau = (3/2)*x0*x0+2*x0*y0+3*y0*y0-2*x0+8*y0
    return mintau

x = 1     # initial guesses for x and y
y = -0.2
f = (3/2)*x*x+2*x*y+3*y*y-2*x+8*y

# track values
xval = np.array(x)
yval = np.array(y)

for j in range(0,100):
    tau = optimize.fmin(tausearch,0.2)
    x = x - tau*(3*x + 2*y - 2)
    y = y - tau*(2*x + 6*y + 8)
    fold = f
    f = (3/2)*x*x+2*x*y+3*y*y-2*x+8*y

    xval = np.append(xval, x)
    yval = np.append(yval, y)

    if abs(f-fold) < 1e-5:
        break
```

Note that an initial guess of $\tau = 0.2$ is used in **optimize.fmin**.

Figure 2.3(a) shows how this algorithm converges to the solution of $\mathbf{Ax} = \mathbf{b}$ for the values of τ computed from (2.3.10). Indeed, the algorithm converges quickly (11 iterations are required) to the correct value of $\mathbf{x} = [2 \quad -2]^T$. Note that if a simple radially symmetric function is considered, then the gradient descent converges in a single iteration since the gradient descent would point directly at the minimum. This figure assumes a line search algorithm to find an optimal value of τ. In particular, the value of τ picked here is optimal in the sense that a given line search is conducted so that the minimum of the gradient direction is picked as the next iteration point. However, this is not a requirement. In fact, one can simply choose a fixed value of τ for stepping forward along the gradient direction. Figure 2.3(b) demonstrates this case for $\tau = 0.1$. This method also converges to the solution, however at a much slower rate. Such a method may be favourable in a case where the steepest descent algorithm *zig-zags* a large amount in trying to make the projective steps orthogonal. This can happen in cases where *long-valley* type structures exist in the function we are trying to minimize.

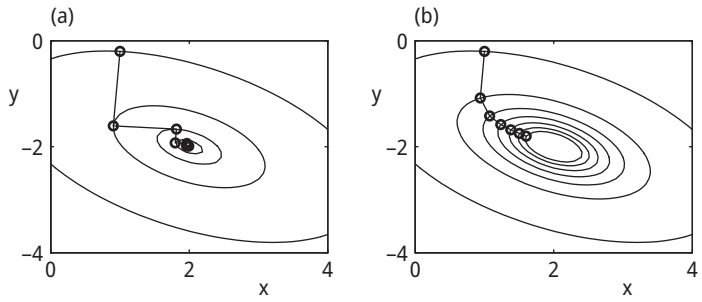


Figure 2.3 Gradient descent algorithm applied to the function (quadratic form) $f = (3/2)x_1^2 + 2x_1x_2 + 3x_2^2 - 2x_1 + 8x_2$. In both panels, the contours are plotted for each successive value (x, y) in the iteration algorithm given the initial guess $(x, y) = (1, -0.2)$. For (a), the optimal τ value is calculated so that each successive gradient in the steepest descent algorithm is orthogonal to the previous step. In (b), a prescribed value of $\tau = 0.1$ is used. In this case, the iteration still converges to the solution but at a much slower rate since the descent steps are not chosen in an optimal way.

Gradient descent is a first example of how to use a quadratic form along with a derivative (gradient) for building an iterative scheme for solving $\mathbf{Ax} = \mathbf{b}$. It can also be used as the basis for an optimization routine. As might be guessed, there are a myriad of improvements to such a scheme. Indeed, the gradient descent algorithm is the core algorithm of advanced iterative solvers such as the bi-conjugate gradient descent method (**bicgstab**) and generalized method of residuals (**gmres**), both of which are essentially improvements on the gradient descent considered here. For a more detailed analysis of these schemes, please see the excellent overview by J. R. Shewchuk [9]. The implementation of **gmres** and **bicgstab** in Python is mentioned in the previous section.

2.4 Eigenvalues, Eigenvectors, and Solvability

Another class of linear systems of equations which are of fundamental importance are known as eigenvalue problems. Unlike the system $\mathbf{Ax} = \mathbf{b}$, which has the single unknown vector $\vec{x}$, eigenvalue problems are of the form

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.4.1}$$

which have the unknowns $\mathbf{x}$ and λ. The values of λ are known as the *eigenvalues* and the corresponding $\mathbf{x}$ are the *eigenvectors*.

Eigenvalue problems often arise from differential equations. Specifically, we consider the example of a linear set of coupled differential equations

$$\frac{d\mathbf{y}}{dt} = \mathbf{Ay}. \tag{2.4.2}$$

By attempting a solution of the form

$$\mathbf{y}(t) = \mathbf{x}\exp(\lambda t), \tag{2.4.3}$$

where all the time-dependence is captured in the exponent, the resulting equation for $\mathbf{x}$ is

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{2.4.4}$$

which is just the eigenvalue problem. Once the full set of eigenvalues and eigenvectors of this equation is found, the solution of the differential equation is written as the linear superposition

$$\vec{y} = c_1\mathbf{x}_1\exp(\lambda_1 t) + c_2\mathbf{x}_2\exp(\lambda_2 t) + \cdots + c_N\mathbf{x}_N\exp(\lambda_N t) \tag{2.4.5}$$

where N is the number of linearly independent solutions to the eigenvalue problem for the matrix $\mathbf{A}$ which is of size $N \times N$. Thus, solving a linear system of differential equations relies on the solution of an associated eigenvalue problem.

The question remains: how are the eigenvalues and eigenvectors found? To consider this problem, we rewrite the eigenvalue problem as

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{I}x \tag{2.4.6}$$

where a multiplication by unity has been performed, i.e. $\mathbf{I}x = x$, where $\mathbf{I}$ is the identity matrix. Moving the right-hand side to the left side of the equation gives

$$\mathbf{A}\mathbf{x} - \lambda\mathbf{I}x = 0. \tag{2.4.7}$$

Factoring out the vector $\mathbf{x}$ then gives the desired result

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}. \tag{2.4.8}$$

Two possibilities now exist.

Option I

The determinant of the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is not zero. If this is true, the matrix is *nonsingular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, exists. The solution to the eigenvalue problem (2.4.8) is then

$$\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{0} \tag{2.4.9}$$

which implies that

$$\mathbf{x} = \mathbf{0}. \tag{2.4.10}$$

This trivial solution could have been guessed from (2.4.8). However, it is not relevant as we require nontrivial solutions for $\mathbf{x}$.

Option II

The determinant of the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is zero. If this is true, the matrix is *singular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, cannot be found. Although there is no longer a guarantee that there is a solution, it is the only scenario which allows for the possibility of $\mathbf{x} \neq 0$. It is this condition which allows for the construction of eigenvalues and eigenvectors. Indeed, we choose the eigenvalues λ so that this condition holds and the matrix is singular.

To illustrate how the eigenvalues and eigenvectors are computed, an example is shown. Consider the 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix}. \tag{2.4.11}$$

This gives the eigenvalue problem

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix} \mathbf{x} = \lambda\mathbf{x} \tag{2.4.12}$$

which when manipulated to the form $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}$ gives

$$\left[\begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right] \mathbf{x} = \begin{pmatrix} 1-\lambda & 3 \\ -1 & 5-\lambda \end{pmatrix} \mathbf{x} = 0. \tag{2.4.13}$$

We now require that the determinant is zero:

$$\det \begin{vmatrix} 1-\lambda & 3 \\ -1 & 5-\lambda \end{vmatrix} = (1-\lambda)(5-\lambda) + 3 = \lambda^2 - 6\lambda + 8 = (\lambda - 2)(\lambda - 4) = 0. \tag{2.4.14}$$

The *characteristic equation* for determining λ in the 2×2 case reduces to finding the roots of a quadratic equation. Specifically, this gives the two eigenvalues

$$\lambda = 2, 4. \tag{2.4.15}$$

For an $N \times N$ matrix, the characteristic equation is an N degree polynomial that has N roots.

The eigenvectors are then found from (2.4.13) as follows:

$$\lambda = 2: \quad \begin{pmatrix} 1-2 & 3 \\ -1 & 5-2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} -1 & 3 \\ -1 & 3 \end{pmatrix} \mathbf{x} = 0. \tag{2.4.16}$$

Given that $\mathbf{x} = (x_1 \;\; x_2)^T$, this leads to the single equation

$$-x_1 + 3x_2 = 0. \tag{2.4.17}$$

This is an underdetermined system of equations. Thus, we have freedom in choosing one of the values. Choosing $x_2 = 1$ gives $x_1 = 3$ and determines the first eigenvector to be

$$\mathbf{x}_1 = \begin{pmatrix} 3 \\ 1 \end{pmatrix}. \tag{2.4.18}$$

The second eigenvector comes from (2.4.13) as follows:

$$\lambda = 4: \quad \begin{pmatrix} 1-4 & 3 \\ -1 & 5-4 \end{pmatrix} \mathbf{x} = \begin{pmatrix} -3 & 3 \\ -1 & 1 \end{pmatrix} \mathbf{x} = 0. \tag{2.4.19}$$

Given that $\mathbf{x} = (x_1 \;\; x_2)^T$, this leads to the single equation

$$-x_1 + x_2 = 0. \tag{2.4.20}$$

This again is an underdetermined system of equations. Thus, we have freedom in choosing one of the values. Choosing $x_2 = 1$ gives $x_1 = 1$ and determines the second eigenvector to be

$$\mathbf{x}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{2.4.21}$$

These results can be found from Python by using the **eig** command. Specifically, the command structure

```
from numpy import linalg
D,V = linalg.eig(A)
```

gives the matrix $\mathbf{V}$ containing the eigenvectors as columns and the matrix $\mathbf{D}$ whose diagonal elements are the corresponding eigenvalues.

For very large matrices, it is often only required to find the largest or smallest eigenvalues. This can easily be done with the **eigs** command. Thus, to determine the K largest in magnitude eigenvalues, the following command is used:

```
sorted_indices = np.argsort(np.abs(D))[::-1]
Dsort = D[sorted_indices]
Vsort =V[:, sorted_indices]
```

Instead of all N eigenvectors and eigenvalues, only the first K will now be returned with the largest magnitude. The option LM denotes the largest magnitude eigenvalues. Often the smallest, largest or smallest real, largest real eigenvalues are required and the **eigs** command allows you to construct these easily.

2.4.1 Matrix powers

Another important operation which can be performed with eigenvalues and eigenvectors is the evaluation of

$$\mathbf{A}^M \tag{2.4.22}$$

where M is a large integer. For large matrices $\mathbf{A}$, this operation is computationally expensive. However, knowing the eigenvalues and eigenvectors of $\mathbf{A}$ allows for a significant reduction in computational expense. Assuming we have all the eigenvalues and eigenvectors of $\mathbf{A}$, then

$$\begin{aligned} \mathbf{A}\mathbf{x}_1 &= \lambda_1 \mathbf{x}_1 \\ \mathbf{A}\mathbf{x}_2 &= \lambda_2 \mathbf{x}_2 \\ &\vdots \\ \mathbf{A}\mathbf{x}_n &= \lambda_n \mathbf{x}_n. \end{aligned}$$

This collection of eigenvalues and eigenvectors gives the matrix system

$$\mathbf{A}\mathbf{S} = \mathbf{S}\mathbf{\Lambda} \tag{2.4.23}$$

where the columns of the matrix $\mathbf{S}$ are the eigenvectors of $\mathbf{A}$,

$$\mathbf{S} = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{pmatrix}, \tag{2.4.24}$$

and $\mathbf{\Lambda}$ is a matrix whose diagonals are the corresponding eigenvalues

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n \end{pmatrix}. \tag{2.4.25}$$

By multiplying (2.4.24) on the right by $\mathbf{S}^{-1}$, the matrix $\mathbf{A}$ can then be rewritten as

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}. \tag{2.4.26}$$

The final observation comes from

$$\mathbf{A}^2 = (\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1})(\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}) = \mathbf{S}\mathbf{\Lambda}^2\mathbf{S}^{-1}. \tag{2.4.27}$$

This then generalizes to

$$\mathbf{A}^M = \mathbf{S}\mathbf{\Lambda}^M\mathbf{S}^{-1} \tag{2.4.28}$$

where the matrix $\mathbf{\Lambda}^M$ is easily calculated as

$$\mathbf{\Lambda}^M = \begin{pmatrix} \lambda_1^M & 0 & \cdots & & 0 \\ 0 & \lambda_2^M & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n^M \end{pmatrix}. \tag{2.4.29}$$

Since raising the diagonal terms to the Mth power is easily accomplished, the matrix $\mathbf{A}^M$ can then be easily calculated by multiplying the three matrices in (2.4.28).

2.4.2 Solvability and the Fredholm-alternative theorem

It is easy to ask under what conditions the system

$$\mathbf{Ax} = \mathbf{b} \tag{2.4.30}$$

can be solved. Aside from requiring the det$\mathbf{A} \neq 0$, we also have a solvability condition on $\mathbf{b}$. Consider the adjoint problem

$$\mathbf{A}^\dagger \mathbf{y} = 0 \tag{2.4.31}$$

where $\mathbf{A}^\dagger = \mathbf{A}^{*T}$ is the adjoint which is the transpose and complex conjugate of the matrix $\mathbf{A}$.

The definition of the adjoint is such that

$$\mathbf{y} \cdot \mathbf{Ax} = \mathbf{A}^\dagger \mathbf{y} \cdot \mathbf{x}. \tag{2.4.32}$$

Since $\mathbf{Ax} = \mathbf{b}$, the left side of the equation reduces to $\mathbf{y} \cdot \mathbf{b}$, while the right side reduces to $\mathbf{0}$ since $\mathbf{A}^\dagger \mathbf{y} = \mathbf{0}$. This then gives the condition

$$\mathbf{y} \cdot \mathbf{b} = \mathbf{0} \tag{2.4.33}$$

which is known as the *Fredholm-alternative* theorem, or a *solvability* condition. In words, the equation states that in order for the system $\mathbf{Ax} = \mathbf{b}$ to be solvable, the right-hand side forcing $\mathbf{b}$ must be orthogonal to the null space of the adjoint operator $\mathbf{A}^\dagger$.

2.5 Eigenvalues and Eigenvectors for Face Recognition

Normally, to motivate the concept of what eigenvalues and eigenvectors mean, examples would be given from some physical and/or engineering system. Often, however, the insight given into the physical system assumes that you actually know the workings of the physical system fairly well. Thus, eigenvalues and eigenvectors often fail to impress students as they should. As a generic statement, it is difficult to refute the claim that if you know the

eigenvectors and eigenvalues of a given system (or matrix), then you know everything there is to know about that system. Given then their importance, some measure of intuition will be developed about their meaning and use.

To motivate the importance of eigenvalues and eigenvectors, they will be considered in the context of the computer vision problem of human face recognition. The approach of using so-called *eigenfaces* for recognition was developed by Sirovich and Kirby [10, 11] and used by Turk and Pentland [12] in face classification. It is considered the first successful example of facial recognition technology. In what follows, some very basic ideas of eigenfaces will be given.

First, consider a set of face images. Figure 2.4 will be our so-called *training set* of images. This will be a highly limited example, as we will consider only faces from four different people representing the political, entertainment, and sports arenas, namely George Clooney, Barack Obama, Margaret Thatcher, and Matt Damon. For each celebrity, five images are considered that are roughly cropped the same way. If you were a computer scientist, you would probably



Figure 2.4 Five image samples of head shots of four different faces including George Clooney (top row), Barack Obama (second row), Margaret Thatcher (third row), and Matt Damon (bottom row). Each image was cropped, turned to greyscale, and resized to 120 vertical and 80 horizontal pixels. All images are public domain, licensed under Creative Commons Attribution-Share Alike 2.0, or licensed under Creative Commons Attribution 3.0.

also make sure to subtract the background from the pictures, make sure all faces are looking directly at the camera without a head tilt, have photos with identical (or normalized) lighting, crop in a more systematic way, etc. However, this is only a rough guide on how the techniques work, and we will not concern ourselves with all of the extras that can make the eigenface methodology perform better.

To begin, all images are imported, turned into greyscale, and resized to the same number of pixels in the vertical (120 pixels) and horizontal (80 pixels) directions. The following code takes an image called **pic.jpg**, converts it into greyscale and turns the image format into double precision numbers for manipulation. Resizing is also accomplished.

```
import matplotlib.image as img
from skimage.color import rgb2gray
from skimage import io
from skimage.transform import resize

Xc=io.imread('pic.jpg')  # load color image
X = rgb2gray(Xc)  # make image black-and-white
Xr= resize(X,(120,80))  # resize to 120 x 80 array
```

This preprocessing is done for each picture to create an image matrix $\mathbf{A}$. A convenient way to view the image is with the **pcolor** command

```
plt.pcolor(np.flipud(Xr),cmap='gray')
plt.xticks([]); plt.yticks([])
```

Note that the **flipud** command flips the image to be right-side-up for visualization. Further, the **imresize** command is part of the image processing toolbox.

To start understanding the face recognition process, we can begin by trying to understand the concept of an *average* face. In particular, if one takes the five images of a single person represented in Fig. 2.4 and averages over them, one would arrive at an average face for that person. Figure 2.5 demonstrates the average face associated with the five images of the four considered celebrities. The average faces are trivially computed by the following Python code:

```
Ave=(A1+A2+A3+A4+A5)/5;
```

where $\mathbf{A1}$ through $\mathbf{A5}$ are the five images of a particular celebrity. Note that although each celebrity image has a different background and is slightly misaligned from the others, the average still represents the basic look of the celebrity. In some sense, this is their *average* face. Interestingly enough, each picture can be thought of as the average plus corrections,



Figure 2.5 Average faces generated from the five images of each celebrity of Fig. 2.4. Despite the varying backgrounds and image misalignments, the images retain the fundamental aspects of the individuals considered.

or variance from the average. And such is the fundamental philosophy of the approach of eigenfaces: there exists an average face and all other faces are simply variances (perhaps small or large) from the average face. Indeed, it is conjectured that each person's variance from the average face is unique so that identification of the face can be made from them. To get a good average face for human beings, or even individuals or select groups of individuals, one would need very large data sets to improve the eigenface methodology.

What the average face actually emphasizes is the common features among a group of pictures, i.e. the pixels that are correlated. This gives a clue on how to pursue a face recognition algorithm: look for the correlations and common features of faces. Alternatively, two faces that are highly uncorrelated are most likely different. Correlations between data sets can be computed using the *covariance*. To do this, all the images in Fig. 2.4 are arranged into a matrix so that

$$\mathbf{B} = \begin{pmatrix} image1 \\ image2 \\ \vdots \\ imageN \end{pmatrix} \tag{2.5.1}$$

where each image has been reshaped into a vector using the command

```
a = resize(A,(1,120*80))
```

The new matrix $\mathbf{B}$ now has individual images that comprise each row. In our case, this will be a matrix with 20 rows with 120×80 columns.

To compute a correlation matrix, it only remains to multiply each row vector by each other row vector. Specifically, the inner product of any two rows gives the correlation between those two images. One can also normalize this if desired, but we will not do so here. Further, most eigenface methodologies subtract the average face of the entire data set so that everything is about the deviation from this average face. Again, this is not necessary to demonstrate the method. To correlate all the pixels and images, i.e. to multiply all rows by all other rows, we can simply compute the correlation matrix

$$C = B^T B \tag{2.5.2}$$

where the superscript T denotes the transpose of the data matrix. This results in a very large correlation matrix (9600×9600).

Our objective is to then compute the eigenvalues and eigenvectors of this matrix in order to gain some insight into the face recognition algorithm. To compute the eigenvalues and eigenvectors, simply apply the code from the last section

```
C = np.matmul(B,B.T)
D,V = linalg.eig(C)
```

where the eigenvalues and eigenvectors are generated, which can then be sorted by the largest magnitude eigenvalues. The matrix $\mathbf{V}$ contains the first eigenvectors as the columns, and the vector $\mathbf{D}$ contains the first 20 eigenvalues as its diagonals. Figure 2.6 shows the first five eigenvectors (reshaped back to matrix form for viewing) of our data matrix. Additionally, the first 20 eigenvalues are demonstrated showing the decrease in their value. The magnitude of the eigenvalues is directly related to how important its eigenvector is in composing the images. In the log plot of Fig. 2.6, the first mode dominates the *energy* content of all the images. This is effectively the average face within the 25 selected images. The other modes are shown as a function of decreasing importance. The idea is that one can reconstruct any of the faces considered as a weighted sum of the eigenvectors shown. All that remains is a demonstration on how the weighting of the eigenvectors is achieved. Given the dominance of the first few eigenvectors and their importance in the weighting, they are often called *principal components*, or a principal component analysis (PCA). As the name implies, one can think of them as the principal quantities (or most important quantities) of interest in the face recognition algorithm. A great deal more will be discussed concerning PCA in the third part of this book on data analysis. What is important here is that the eigenvalue/eigenvector decomposition of the matrix highlights the most essential features of the data matrix and also ranks, or prioritizes, through the eigenvalues the importance of each eigenvector.

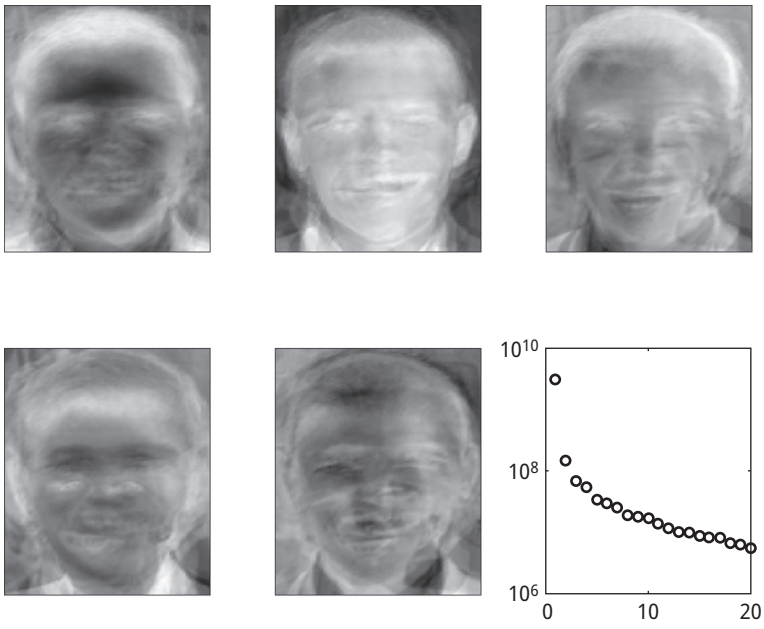


Figure 2.6 The first five eigenvectors (or eigenface components) of the correlation matrix $\mathbf{C}$ arranged from top left to bottom right. In the bottom right panel, the eigenvalues of the first 20 eigenvalues are shown. Note the dominance of the first mode on the log scale which suggests it captures the average face in the data matrix. The additional eigenvectors are added in a weighted fashion to the average mode in order to produce the individual faces (or variances) of each image.

To represent an individual in terms of the eigenvectors of the full data matrix $\mathbf{C}$, it remains to understand the weighting of each image on the eigenvectors. This can be done by simply computing the inner product of each image on each eigenvector. This can be done very simply with the Python command

```
proj_a = np.matmul(a,V)
```

where it should be recalled that $\mathbf{a}$ is the reshaped picture in vector form of a specific image. Multiplying by the eigenvector matrix $\mathbf{V}$ gives a vector of inner products of the image onto each eigenvector. This operation can be performed for the average image (see Fig. 2.5) of the four celebrities considered in Fig. 2.4. This gives a *unique* representation of each celebrity on the eigenvectors. Figure 2.7 shows the projection coefficients of the four average faces of Fig. 2.5. Note that each celebrity has a different set of coefficients in this representation. It is these differences that can be used as the basis for face recognition.

To demonstrate how the eigenvectors can be used in practice, we consider reconstruction of a *new* image using the eigenvectors formed from our *training set*. Figure 2.8 shows three new images that are not part of the training set. In the first, a new picture of Margaret Thatcher (Fig. 2.8 top row) is introduced and projected onto the first 20 eigenvalues. The

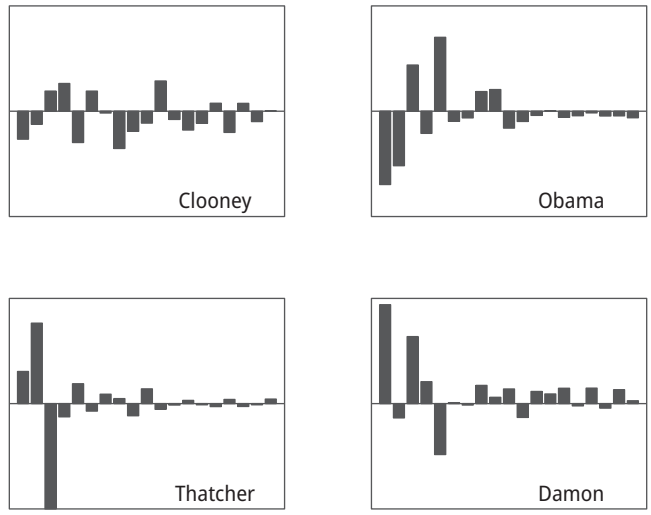


Figure 2.7 Projection of the average faces Fig. 2.5 onto the eigenvector space computed from the full correlation matrix $\mathbf{C}$. Note that each celebrity has a different, and hopefully unique, set of coefficients that can be used for face recognition.

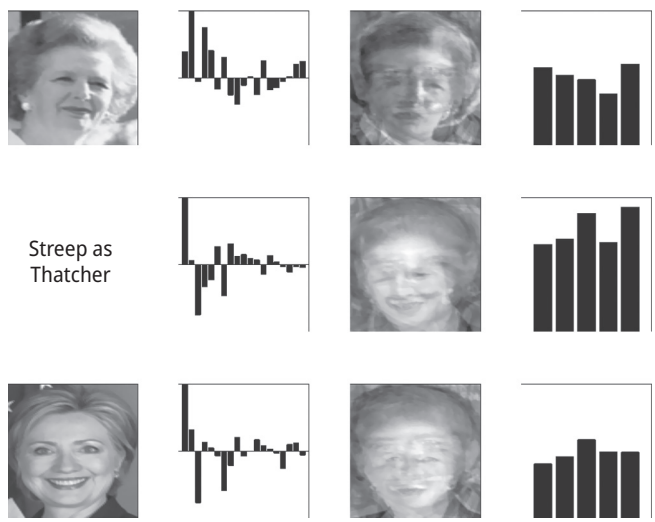


Figure 2.8 Projection of three new images (left panels) onto the eigenvector space created from Fig. 2.4. The coefficients of the projection are shown in the second columns of data panels while the reconstruction using 20 eigenvectors is shown in the third column of data. The fourth column shows the distance in the coefficient measure (2.5.3) for each new picture against the original five Margaret Thatcher images. The new Margaret Thatcher image is able to produce a reasonable reconstruction while Meryl Streep (as Thatcher) and Clinton both perform poorly under reconstruction. Images of Thatcher and Clinton are both public domain.

weighting coefficients are computed and a 20-mode representation is then constructed in the third panel of the top row. This shows that some semblance of Margaret Thatcher can indeed be reconstructed from our training set, which included five different pictures of Margaret Thatcher. In the last panel of the top row, the difference between the projection vector of the new image and the projection vector of the original five Margaret Thatcher images is given. Thus, the following quantity is computed:

$$E_j = \frac{\|c_j - c_{\text{new}}\|}{\|c_j\|} \tag{2.5.3}$$

where the double bar denotes the L^2 norm, c_j is the projection of the jth Margaret Thatcher image in the original set of Fig. 2.4 and c_{new} is the new image presented. In Python, this can be accomplished with

```
E=np.linalg.norm(proj_a-proj_new)/np.linalg.norm(proj_a);
```

This gives some measure of the difference between the new images and the original images already used in the correlation matrices. The conjecture is that if the image is similar to those in the training set, then this error, or difference, should be small. In the second row of Fig. 2.8, the results are reproduced with a new picture (not shown) of Meryl Streep who plays Margaret Thatcher in the movie *Iron Lady* (2012). We can compare her Margaret Thatcher look to the real Margaret Thatcher to conclude that with a 20-mode reconstruction (second row, third panel), she does not look much like Margaret Thatcher from the perspective of the training set used. The last panel in the second row shows the distance measured from the new picture to the five originals in terms of the coefficients of the first 20 eigenvalues. Note that if a totally different person is projected onto the data set (Hillary Clinton in the last row of Fig. 2.8), then no recognizable face is produced in the 20-mode reconstruction. Interestingly, however, Hillary Clinton does appear to be closer in the coefficient space to Margaret Thatcher than Meryl Streep.

Finally, the author wishes to see how he might project onto this data set. This projection is shown in Fig. 2.9. In the top row, the original picture is shown along with the projection onto the coefficients of the eigenvectors and the reconstruction using 20 eigenvectors. What seems to be the case is that the author kind of looks like a cross between Barack Obama and Matt Damon. In fact, when comparing in the eigenvector coefficient space against all the images of Fig. 2.4, the distance calculation (2.5.3) shows the author to be quite similar to Obama/Damon in a couple of his pictures. The author was hoping to look perhaps more like George Clooney.

As a final note, the technique used in creating eigenfaces and using them for recognition is also used outside of facial recognition. This technique is also used for handwriting analysis, lip reading, voice recognition, sign language/hand gestures interpretation, and medical imaging analysis. Therefore, some do not use the term eigenface, but prefer to use *eigenimage*.

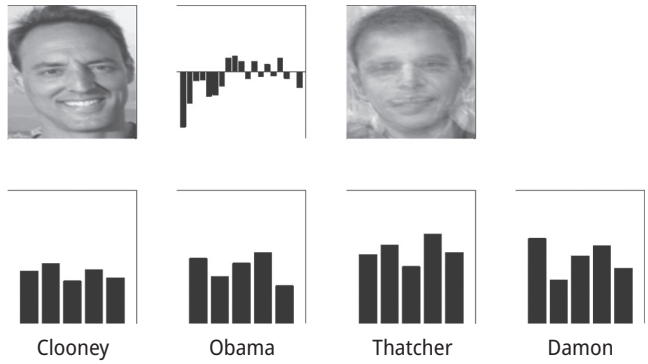


Figure 2.9 Projection of the author onto the eigenvector space created from Fig. 2.4. The coefficients of the projection are shown in the second panel of the top row while the reconstruction using 20 eigenvectors is shown in the third column. Seems like there is some resemblance to Obama and Damon. The bottom row shows the distance in the coefficient measure (2.5.3) for the author's picture against the original 25 images, showing that the author looks most like Obama than the others.

2.6 Nonlinear Systems

The preceding sections deal with a variety of techniques to solve linear systems of equations. Provided the matrix $\mathbf{A}$ is nonsingular, there is one unique solution to be found. Often, however, it is necessary to consider a *nonlinear* system of equations. This presents a significant challenge since a nonlinear system may have no solutions, one solution, five solutions, or an infinite number of solutions. Unfortunately, there is no general theorem concerning nonlinear systems which can narrow down the possibilities. Thus, even if we are able to find a solution to a nonlinear system, it may be simply one of many. Furthermore, it may not be a solution we are particularly interested in.

To help illustrate the concept of nonlinear systems, we can once again consider the Newton–Raphson method of Section 1.3. In this method, the roots of a function $f(x)$ are determined by the Newton–Raphson iteration method. As a simple example, Fig. 2.10 illustrates a simple cubic function which has either one or three roots. For an initial guess, the Newton solver will converge to the one root for the example shown in Fig. 2.10(a). However, Fig. 2.10(b) shows that the initial guess is critical in determining which root the solution finds. In this simple example, it is easy to graphically see there are either one or three roots. For more complicated systems of nonlinear equations, we in general will not be able to visualize the number of roots or how to guess. This poses a significant problem for the Newton iteration method.

The Newton method can be generalized for solving systems of nonlinear equations. The details will not be discussed here, but the Newton iteration scheme is similar to that developed for the single-function case. Given a system:

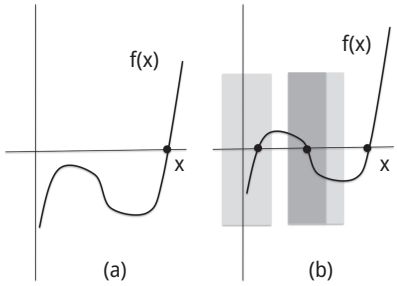


Figure 2.10 Two example cubic functions and their roots. In (a), there is only a single root to converge on with a Newton iteration routine. Aside from guessing a location which has $f'(x) = 0$, the solution will always converge to this root. In (b), the cubic has three roots. Guessing in the dark grey region will converge to the middle root. Guessing in the light grey regions will converge to the left root. Guessing anywhere else will converge to the right root. Thus, the initial guess is critical to finding a specific root location.

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, x_3, \ldots, x_N) \\ f_2(x_1, x_2, x_3, \ldots, x_N) \\ \vdots \\ f_N(x_1, x_2, x_3, \ldots, x_N) \end{bmatrix} = 0, \tag{2.6.1}$$

the iteration scheme is

$$\mathbf{x}_{\mathrm{n}+1} = \mathbf{x}_{\mathrm{n}} + \Delta\mathbf{x}_{\mathrm{n}} \tag{2.6.2}$$

where

$$\mathbf{J}(\mathbf{x}_{\mathrm{n}})\Delta\mathbf{x}_{\mathrm{n}} = -\mathbf{F}(\mathbf{x}_{\mathrm{n}}) \tag{2.6.3}$$

and $\mathbf{J}(\mathbf{x}_{\mathrm{n}})$ is the Jacobian matrix

$$\mathbf{J}(\mathbf{x}_{\mathrm{n}}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{bmatrix}. \tag{2.6.4}$$

This algorithm relies on initially guessing values for $x_1, x_2, \ldots, x_N$. As before, the algorithm is guaranteed to converge for an initial iteration value, which is sufficiently close to a solution of (2.6.1). Thus, a good initial guess is critical to its success. Further, the determinant of the Jacobian cannot equal zero, $\det \mathbf{J}(\mathbf{x}_n) \neq 0$, in order for the algorithm to work. Indeed, significant problems can occur any time the determinant is nearly zero. A simple check of the Jacobian can be made with the condition number command: **cond(J)**. If this condition

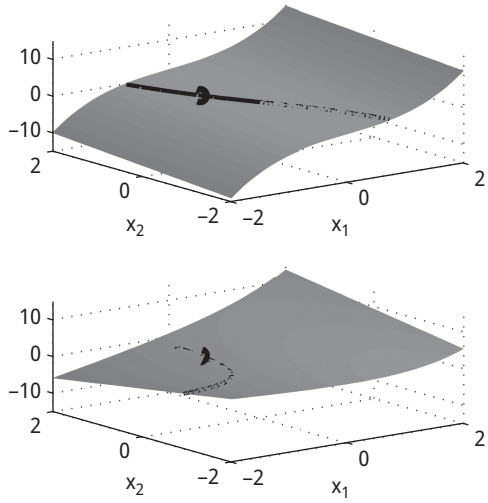


Figure 2.11 Plot of the two surfaces: $f_1(x_1, x_2) = 2x_1 + x_2 + x_1^3$ (top) and $f_2(x_1, x_2) = x_1 + x_1x_2 + \exp(x_1)$ (bottom). The solid lines show where the crossing at zero occurs for both surfaces. The dot is the intersection of zero surfaces and the solution of the 2×2 system (2.6.5).

number is large, i.e. greater than 10^6, then problems are likely to occur in the iteration algorithm. This is equivalent to having an almost zero derivative in the Newton algorithm displayed in Fig. 2.4. Specifically, a large condition number or nearly zero derivative will move the iteration far away from the objective fixed point.

To show the practical implementation of this method, consider the following nonlinear system of equations:

$$f_1(x_1, x_2) = 2x_1 + x_2 + x_1^3 = 0 \tag{2.6.5a}$$

$$f_2(x_1, x_2) = x_1 + x_1x_2 + \exp(x_1) = 0. \tag{2.6.5b}$$

Figure 2.11 shows a surface plot of the functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ with bolded lines for demarcation of the lines where these functions are zero. The intersection of the zero lines of f_1 and f_2 are the solutions of the example 2×2 system (2.6.5).

To apply the Newton iteration algorithm, the Jacobian must first be calculated. The partial derivatives required for the Jacobian are first evaluated:

$$\frac{\partial f_1}{\partial x_1} = 2 + 3x_1^2 \tag{2.6.6a}$$

$$\frac{\partial f_1}{\partial x_2} = 1 \tag{2.6.6b}$$

$$\frac{\partial f_2}{\partial x_1} = 1 + x_2 + \exp(x_1) \tag{2.6.6c}$$

$$\frac{\partial f_2}{\partial x_2} = x_1 \tag{2.6.6d}$$

and the Jacobian is constructed

$$\mathbf{J} = \begin{pmatrix} 2 + 3x_1^2 & 1 \\ 1 + x_2 + \exp(x_1) & x_1 \end{pmatrix}. \tag{2.6.7}$$

A simple iteration procedure can then be developed for Python. The following Python code illustrates the ease of implementation of this algorithm.

```
x = np.array([0, 0], dtype=float)

for j in range(1000):
    J = np.array([[2 + 3*x[0]**2, 1],
                  [1 + x[1] + np.exp(x[0]), x[0]]])
    f = np.array([2*x[0] + x[1] + x[0]**3,
                  x[0] + x[0]*x[1] + np.exp(x[0])])

    if np.linalg.norm(f) < 10**(-6):
        break

    df = -np.linalg.solve(J, f)
    x = x + df
```

In this algorithm, the initial guess for the solution was taken to be $(x_1, x_2) = (0, 0)$. The Jacobian is updated at each iteration with the current values of x_1 and x_2. Table 2.4 shows the convergence to an accuracy of 10^{-6} of this simple algorithm to the solution of (2.6.5). It only requires four iterations to calculate the solution of (2.6.5). Recall, however, that if there were multiple roots in such a system, then the difficulties would arise from trying to guess initial solutions so that one can find all the solution roots.

The built-in Python command which solves nonlinear system of equations is **fsolve**. The **fsolve** algorithm is a more sophisticated version of the Newton method presented here. As with any iteration scheme, it requires that the user provide an initial starting guess for the solution. The basic command structure is as follows:

Table 2.4 Convergence of Newton iteration algorithm to the roots of (2.6.5). A solution, to an accuracy of 10^{-6} is achieved, with only four iterations with a starting guess of $(x_1, x_2) = (0, 0)$.

Iteration	x_1	x_2
1	−0.50000000	1.00000000
2	−0.38547888	0.81006693
3	−0.37830667	0.81069593
4	−0.37830658	0.81075482

```
from scipy.optimize import fsolve

def system(x):
    f = np.array([2*x[0] + x[1] + x[0]**3,
                  x[0] + x[0]*x[1] + np.exp(x[0])])
    return f

x_initial_guess = np.array([0, 0])
x_solution = fsolve(system, x_initial_guess)
```

where the vector $[0\ 0]$ is the initial guess values for x_1 and x_2. This command calls upon the function **system.m** that contains the function whose roots are to be determined.
Execution of the **fsolve** command with this function **system.m** once again yields the solution to the 2×2 nonlinear system of equations (2.6.5).

A nice feature of the **fsolve** command is that it has many options concerning the search algorithm and its accuracy settings. Thus, one can search for more or less accurate solutions as well as set a limit on the number of iterations allowed in the **fsolve** function call.

Numerical Differentiation and Integration

3

3.1 Numerical Differentiation 69
3.1.1 Round-off and optimal step-size 73
3.2 Numerical Integration 75
3.2.1 Newton–Cotes formulas 76
3.2.2 Composite rules 78
3.2.3 Recursive improvement of accuracy 78
3.3 Implementation of Differentiation and Integration 79
3.3.1 Differentiation 80
3.3.2 Second-order accurate derivative 81
3.3.3 Fourth-order accurate derivative 81
3.3.4 Integration 83
3.4 Differentiation of Noisy Data 85
3.4.1 Spline smoothing 85
3.4.2 Savitzky–Golay Filtering 87
3.4.3 Alternative Methods 88
3.4.4 PyNumDiff 89

Differentiation and integration form the backbone of the mathematical techniques required to describe and analyse physical systems. These two mathematical concepts describe how certain quantities of interest change with respect to either space and time or both. Understanding how to evaluate these quantities numerically is essential to understanding systems beyond the scope of analytic methods.

3.1 Numerical Differentiation

Given a set of data or a function, it may be useful to differentiate the quantity considered in order to determine a physically relevant property. For instance, given a set of data that represents the position of a particle as a function of time, the derivative and second derivative give the velocity and acceleration, respectively. From calculus, the definition of the derivative is given by

$$\frac{df(t)}{dt} = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}. \tag{3.1.1}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0004

Since the derivative is the slope, the formula on the right is nothing more than a rise-over-run formula for the slope. The general idea of calculus is that as $\Delta t \to 0$, the rise-over-run gives the instantaneous slope. Numerically, this means that if we take Δt sufficiently small, then the approximation should be fairly accurate. To quantify and control the error associated with approximating the derivative, we make use of *Taylor series* expansions.

To see how the Taylor expansions are useful, consider the following two Taylor series:

$$f(t+\Delta t) = f(t) + \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2 f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3 f(c_1)}{dt^3} \tag{3.1.2a}$$

$$f(t-\Delta t) = f(t) - \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2 f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3 f(c_2)}{dt^3} \tag{3.1.2b}$$

where $c_1 \in [t, t+\Delta t]$ and $c_2 \in [t, t-\Delta t]$. Subtracting these two expressions gives

$$f(t+\Delta t) - f(t-\Delta t) = 2\Delta t \frac{df(t)}{dt} + \frac{\Delta t^3}{3!}\left(\frac{d^3 f(c_1)}{dt^3} + \frac{d^3 f(c_2)}{dt^3}\right). \tag{3.1.3}$$

By using the intermediate value theorem of calculus, we find $f'''(c) = (f'''(c_1) + f'''(c_2))/2$. Upon dividing the above expression by $2\Delta t$ and rearranging, we find the following expression for the first derivative:

$$\frac{df(t)}{dt} = \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3 f(c)}{dt^3} \tag{3.1.4}$$

where the last term is the truncation error associated with the approximation of the first derivative using this particular Taylor series generated expression. Note that the truncation error in this case is $O(\Delta t^2)$. Figure 3.1 demonstrates graphically the numerical procedure and approximation for the second-order slope formula. Here, nearest neighbour points are used to calculate the slope.

We could improve on this by continuing our Taylor expansion and truncating it at higher orders in Δt. This would lead to higher accuracy schemes. Specifically, by truncating at $O(\Delta t^5)$, we would have

$$\begin{aligned} f(t+\Delta t) = f(t) &+ \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2 f(t)}{dt^2} \\ &+ \frac{\Delta t^3}{3!}\frac{d^3 f(t)}{dt^3} + \frac{\Delta t^4}{4!}\frac{d^4 f(t)}{dt^4} + \frac{\Delta t^5}{5!}\frac{d^5 f(c_1)}{dt^5} \end{aligned} \tag{3.1.5a}$$

$$\begin{aligned} f(t-\Delta t) = f(t) &- \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2 f(t)}{dt^2} \\ &- \frac{\Delta t^3}{3!}\frac{d^3 f(t)}{dt^3} + \frac{\Delta t^4}{4!}\frac{d^4 f(t)}{dt^4} - \frac{\Delta t^5}{5!}\frac{d^5 f(c_2)}{dt^5} \end{aligned} \tag{3.1.5b}$$

where $c_1 \in [t, t+\Delta t]$ and $c_2 \in [t, t-\Delta t]$. Again subtracting these two expressions gives

$$f(t+\Delta t) - f(t-\Delta t) = 2\Delta t \frac{df(t)}{dt} + \frac{2\Delta t^3}{3!}\frac{d^3 f(t)}{dt^3} + \frac{\Delta t^5}{5!}\left(\frac{d^5 f(c_1)}{dt^5} + \frac{d^5 f(c_2)}{dt^5}\right). \tag{3.1.6}$$

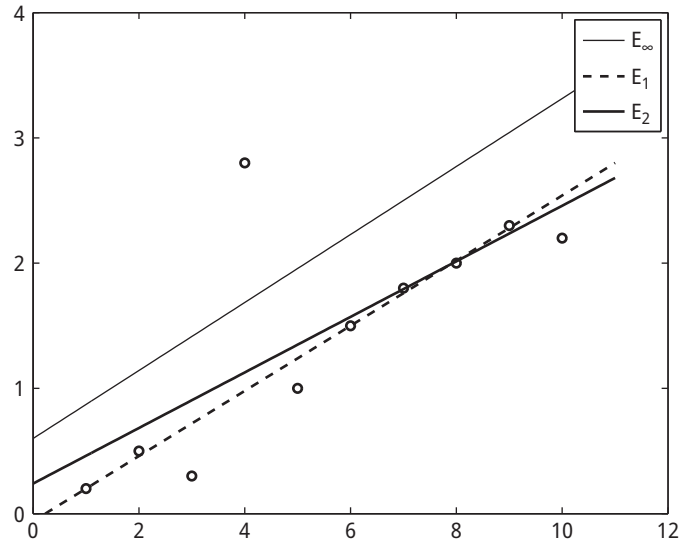


Figure 3.1 Graphical representation of the second-order accurate method for calculating the derivative with finite differences. The slope is simply rise over run, where the nearest neighbours are used to determine both quantities. The specific function considered is $y = -\cos(t)$ with the derivative being calculated at $t = 3$ with $\Delta t = 0.5$.

In this approximation, there is a third derivative term left over, which needs to be removed. By using two additional points to approximate the derivative, this term can be removed. Thus, we use the two additional points $f(t + 2\Delta t)$ and $f(t - 2\Delta t)$. Upon replacing Δt by $2\Delta t$ in (3.1.6), we find

$$f(t+2\Delta t)-f(t-2\Delta t) = 4\Delta t\frac{df(t)}{dt} + \frac{16\Delta t^3}{3!}\frac{d^3f(t)}{dt^3} + \frac{32\Delta t^5}{5!}\left(\frac{d^5f(c_3)}{dt^5} + \frac{d^5f(c_4)}{dt^5}\right) \quad (3.1.7)$$

where $c_3 \in [t, t + 2\Delta t]$ and $c_4 \in [t, t - 2\Delta t]$. By multiplying (3.1.6) by 8 and subtracting (3.1.7) and using the intermediate value theorem on the truncation terms twice, we find the expression:

$$\frac{df(t)}{dt} = \frac{-f(t + 2\Delta t) + 8f(t + \Delta t) - 8f(t - \Delta t) + f(t - 2\Delta t)}{12\Delta t} + \frac{\Delta t^4}{30} f^{(5)}(c) \quad (3.1.8)$$

where $f^{(5)}$ is the fifth derivative, and the truncation is of $O(\Delta t^4)$.

Approximating higher derivatives works in a similar fashion. By starting with the pair of equations (3.1.2a) and adding, this gives the result

$$f(t + \Delta t) + f(t - \Delta t) = 2\, f(t) + \Delta t^2 \frac{d^2 f(t)}{dt^2} + \frac{\Delta t^4}{4!}\left(\frac{d^4 f(c_1)}{dt^4} + \frac{d^4 f(c_2)}{dt^4}\right). \quad (3.1.9)$$

Table 3.1 Second-order accurate centre-difference formulas.

$O(\Delta t^2)$ centre-difference schemes
$f'(t) = [f(t+\Delta t) - f(t-\Delta t)]/2\Delta t$
$f''(t) = [f(t+\Delta t) - 2f(t) + f(t-\Delta t)]/\Delta t^2$
$f'''(t) = [f(t+2\Delta t) - 2f(t+\Delta t) + 2f(t-\Delta t) - f(t-2\Delta t)]/2\Delta t^3$
$f''''(t) = [f(t+2\Delta t) - 4f(t+\Delta t) + 6f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/\Delta t^4$

Table 3.2 Fourth-order accurate centre-difference formulas.

$O(\Delta t^4)$ centre-difference schemes
$f'(t) = [-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)]/12\Delta t$
$f''(t) = [-f(t+2\Delta t) + 16f(t+\Delta t) - 30f(t) + 16f(t-\Delta t) - f(t-2\Delta t)]/12\Delta t^2$
$f'''(t) = [-f(t+3\Delta t) + 8f(t+2\Delta t) - 13f(t+\Delta t)$ $+13f(t-\Delta t) - 8f(t-2\Delta t) + f(t-3\Delta t)]/8\Delta t^3$
$f''''(t) = [-f(t+3\Delta t) + 12f(t+2\Delta t) - 39f(t+\Delta t) + 56f(t)$ $-39f(t-\Delta t) + 12f(t-2\Delta t) - f(t-3\Delta t)]/6\Delta t^4$

Table 3.3 Second-order accurate forward- and backward-difference formulas.

$O(\Delta t^2)$ forward- and backward-difference schemes
$f'(t) = [-3f(t) + 4f(t+\Delta t) - f(t+2\Delta t)]/2\Delta t$
$f'(t) = [3f(t) - 4f(t-\Delta t) + f(t-2\Delta t)]/2\Delta t$
$f''(t) = [2f(t) - 5f(t+\Delta t) + 4f(t+2\Delta t) - f(t+3\Delta t)]/\Delta t^2$
$f''(t) = [2f(t) - 5f(t-\Delta t) + 4f(t-2\Delta t) - f(t-3\Delta t)]/\Delta t^2$

By rearranging and solving for the second derivative, the $O(\Delta t^2)$ accurate expression is derived

$$\frac{d^2 f(t)}{dt^2} = \frac{f(t+\Delta t) - 2f(t) + f(t-\Delta t)}{\Delta t^2} + O(\Delta t^2) \tag{3.1.10}$$

where the truncation error is of $O(\Delta t^2)$ and is found again by the intermediate value theorem to be $-(\Delta t^2/12) f''''(c)$. This process can be continued to find any arbitrary derivative. Thus, we could also approximate the third, fourth, and higher derivatives using this technique. It is also possible to generate backward- and forward-difference schemes by using points only behind or in front of the current point, respectively. Tables 3.1–3.3 summarize the second-order and fourth-order central difference schemes along with the forward- and backward-difference formulas that are accurate to second order.

A final remark is in order concerning these differentiation schemes. The central difference schemes are an excellent method for generating the values of the derivative in the interior points of a data set. However, at the end points, forward- and backward-difference methods

must be used since they do not have neighbouring points to the left and right, respectively. Thus, special care must be taken at the end points of any computational domain.

It may be tempting to deduce from the difference formulas that as $\Delta t \to 0$, the accuracy only improves in these computational methods. However, this line of reasoning completely neglects the second source of error in evaluating derivatives: numerical round-off.

3.1.1 Round-off and optimal step-size

An unavoidable consequence of working with numerical computations is round-off error. When working with most computations, *double precision* numbers are used. This allows for 16-digit accuracy in the representation of a given number. This round-off has a significant impact upon numerical computations and the issue of time-stepping.

As an example of the impact of round-off, we consider the approximation to the derivative

$$\frac{dy}{dt} = \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} + \epsilon(y(t), \Delta t) \tag{3.1.11}$$

where $\epsilon(y(t), \Delta t)$ measures the truncation error. Upon evaluating this expression in the computer, round-off error occurs so that

$$y(t+\Delta t) = Y(t+\Delta t) + e(t+\Delta t) \tag{3.1.12a}$$

$$y(t-\Delta t) = Y(t-\Delta t) + e(t-\Delta t)\,, \tag{3.1.12b}$$

where $Y(\cdot)$ is the approximated value given by the computer and $e(\cdot)$ measures the error from the true value $y(t)$. Thus, the combined error between the round-off and truncation gives the following expression for the derivative:

$$\frac{dy}{dt} = \frac{Y(t+\Delta t) - Y(t-\Delta t)}{2\Delta t} + E(y(t), \Delta t) \tag{3.1.13}$$

where the total error, E, is the combination of round-off and truncation such that

$$E = E_{\text{round}} + E_{\text{trunc}} = \frac{e(t+\Delta t) - e(t-\Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3y(c)}{dt^3}. \tag{3.1.14}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of the round-off error and the value of the second derivative to be

$$|e(t+\Delta t)| \leq e_r \tag{3.1.15a}$$

$$|-e(t-\Delta t)| \leq e_r \tag{3.1.15b}$$

$$M = \max_{c\in[t_n, t_{n+1}]} \left\{ \left| \frac{d^3y(c)}{dt^3} \right| \right\}. \tag{3.1.15c}$$

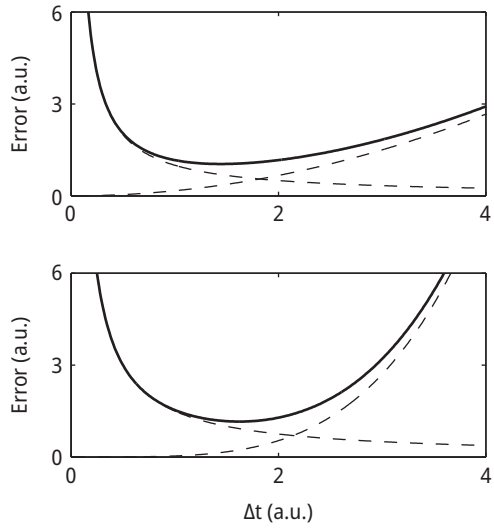


Figure 3.2 Graphical representation of the error, which is made up of two components (dotted lines): numerical round-off and truncation. The total error in arbitrary units is shown for both a second-order scheme (top panel) and a fourth-order scheme (bottom panel). For convenience, we have taken $M = 1$ and $e_r = 1$. Note the dominance of numerical round-off as $\Delta t \to 0$.

This then gives the maximum error to be

$$|E| \leq \frac{e_r + e_r}{2\Delta t} + \frac{\Delta t^2}{6} M = \frac{e_r}{\Delta t} + \frac{\Delta t^2 M}{6}. \tag{3.1.16}$$

Note that as Δt gets large, the error grows linearly due to the truncation error. However, as Δt decreases to zero, the error is dominated by round-off, which grows like $1/\Delta t$. The error as a function of the step-size Δt for this second-order scheme is represented in Fig. 3.2. Note the dominance of the error on numerical round-off as $\Delta t \to 0$.

To minimize the error, we require that $\partial |E| / \partial(\Delta t) = 0$. Calculating this derivative gives

$$\frac{\partial |E|}{\partial(\Delta t)} = -\frac{e_r}{\Delta t^2} + \frac{\Delta t M}{3} = 0\,, \tag{3.1.17}$$

so that

$$\Delta t = \left(\frac{3 e_r}{M} \right)^{1/3}. \tag{3.1.18}$$

This gives the step-size resulting in a minimum error. Thus, the smallest step-size is not necessarily the most accurate. Rather, a balance between round-off error and truncation error is achieved to obtain the optimal step-size. For $e_r \approx 10^{-16}$, the optimal $\Delta t \approx 10^{-5}$. Below this value of Δt, numerical round-off begins to dominate the error.

A similar procedure can be carried out for evaluating the optimal step-size associated with the $O(\Delta t^4)$ accurate scheme for the first derivative. In this case,

$$\frac{dy}{dt} = \frac{-f(t+2\Delta t) + 8f(t+\Delta t) - 8f(t-\Delta t) + f(t-2\Delta t)}{12\Delta t} + E(y(t), \Delta t) \tag{3.1.19}$$

where the total error, E, is the combination of round-off and truncation such that

$$E = \frac{-e(t+2\Delta t) + 8e(t+\Delta t) - 8e(t-\Delta t) + e(t-2\Delta t)}{12\Delta t} + \frac{\Delta t^4}{30}\frac{d^5 y(c)}{dt^5}. \tag{3.1.20}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off to e as before and set $M = \max\{|y''''(c)|\}$. This then gives the maximum error to be

$$|E| \leq \frac{3e_r}{2\Delta t} + \frac{\Delta t^4 M}{30}. \tag{3.1.21}$$

Note that as Δt gets large, the error grows like a quartic due to the truncation error. However, as Δt decreases to zero, the error is again dominated by round-off, which grows like $1/\Delta t$. The error as a function of the step-size Δt for this fourth-order scheme is represented in Fig. 3.2. Note the dominance of the error on numerical round-off as $\Delta t \to 0$.

To minimize the error, we require that $\partial|E|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\Delta t = \left(\frac{45e_r}{4M}\right)^{1/5}. \tag{3.1.22}$$

Thus, in this case, the optimal step $\Delta t \approx 10^{-3}$. This shows that the error can be quickly dominated by numerical round-off, if one is not careful to take this significant effect into account.

3.2 Numerical Integration

Numerical integration simply calculates the area under a given curve. The basic ideas for performing such an operation come from the definition of integration

$$\int_a^b f(x)dx = \lim_{h\to 0}\sum_{j=0}^{N} f(x_j)h \tag{3.2.1}$$

where $b - a = Nh$. Thus, the area under the curve, from the calculus standpoint, is thought of as a limiting process of summing up an ever-increasing number of rectangles. This process is known as numerical quadrature. Specifically, any sum can be represented as follows

$$Q[f] = \sum_{j=0}^{N} w_k f(x_k) = w_0 f(x_0) + w_1 f(x_1) + \cdots + w_N f(x_N) \tag{3.2.2}$$

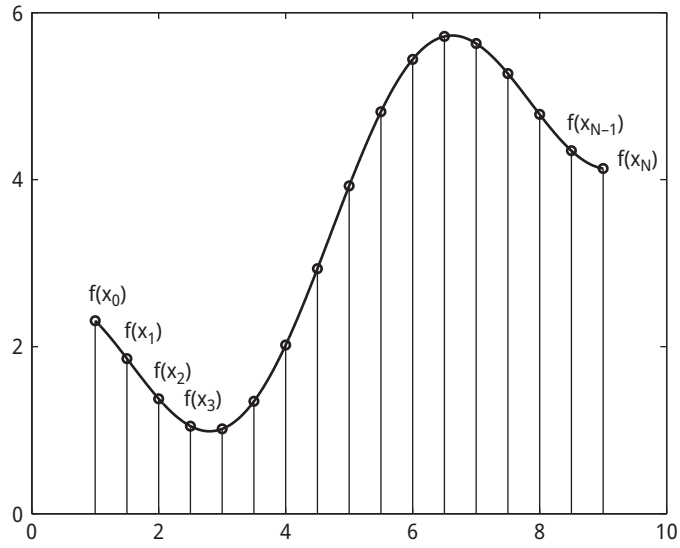


Figure 3.3 Graphical representation of the integration process. The integration interval is broken up into a finite set of points. A quadrature rule then determines how to sum up the area of a finite number of rectangles.

where $a = x_0 < x_1 < x_2 < \cdots < x_N = b$. Thus, the integral is evaluated as

$$\int_a^b f(x)dx = Q[f] + E[f] \tag{3.2.3}$$

where the term $E[f]$ is the error in approximating the integral by the quadrature sum (3.2.2). Typically, the error $E[f]$ is due to truncation error. To integrate, use will be made of polynomial fits to the y-values $f(x_j)$. Thus, we assume the function $f(x)$ can be approximated by a polynomial

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{3.2.4}$$

where the truncation error in this case is proportional to the $(n+1)$th derivative $E[f] = Af^{(n+1)}(c)$ and A is a constant. This process of polynomial fitting the data gives the *Newton–Cotes formulas*. Figure 3.3 gives the standard representation of the integration process and the division of the integration interval into a finite set of integration intervals.

3.2.1 Newton–Cotes formulas

The following integration approximations result from using a polynomial fit through the data to be integrated. It is assumed that

$$x_k = x_0 + hk \qquad\qquad f_k = f(x_k). \tag{3.2.5}$$

This gives the following integration algorithms:

$$\text{Trapezoid rule } \int_{x_0}^{x_1} f(x)dx = \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12} f''(c) \tag{3.2.6a}$$

$$\text{Simpson's rule } \int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90} f''''(c) \tag{3.2.6b}$$

$$\text{Simpson's 3/8 rule } \int_{x_0}^{x_3} f(x)dx = \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80} f''''(c) \tag{3.2.6c}$$

$$\text{Boole's rule } \int_{x_0}^{x_4} f(x)dx = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945} f^{(6)}(c). \tag{3.2.6d}$$

These algorithms have varying degrees of accuracy. Specifically, they are $O(h^2)$, $O(h^4)$, $O(h^4)$, and $O(h^6)$ accurate schemes, respectively. The accuracy condition is determined from the truncation terms of the polynomial fit. Note that the *trapezoid rule* uses a sum of simple trapezoids to approximate the integral. *Simpson's rule* fits a quadratic curve through three points and calculates the area under the quadratic curve. *Simpson's 3/8 rule* uses four points and a cubic polynomial to evaluate the area, while *Boole's rule* uses five points and a quartic polynomial fit to generate an evaluation of the integral. Figure 3.4 represents graphically the different quadrature rules and its approximation (dashed line) to the actual function.

The derivation of these integration rules follows from simple polynomial fits through a specified number of data points. To derive Simpson's rule, consider a second-degree Lagrange polynomial through the three points (x_0, f_0), (x_1, f_1), and (x_2, f_2):

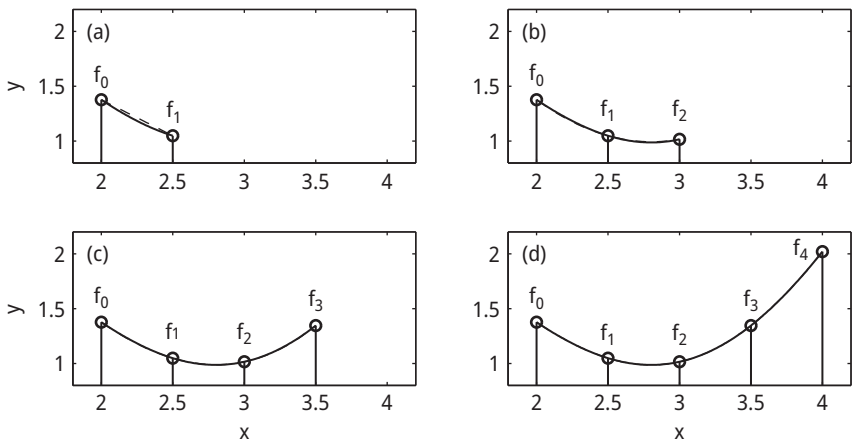


Figure 3.4 Graphical representation of the integration process for the quadrature rules in (3.2.6d). The quadrature rules represented are (a) trapezoid rule, (b) Simpson's rule, (c) Simpson's 3/8 rule, and (d) Boole's rule. In addition to the specific function considered, the dotted line represents the polynomial fit to the actual function for the four different integration rules. Note that the dashed line and function line are essentially identical for (c) and (d).

$$p_2(x) = f_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}. \quad (3.2.7)$$

This quadratic fit is derived by using Lagrange coefficients. The truncation error could also be included, but we neglect it for the present purposes. By plugging in (3.2.7) into the integral

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} p_2(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2). \quad (3.2.8)$$

The integral calculation is easily performed since it only involves integrating powers of x^2 or less. Evaluating at the limits then causes many terms to cancel and drop out. Thus, Simpson's rule is recovered. The trapezoid rule, Simpson's 3/8 rule and Boole's rule are all derived in a similar fashion. To make connection with the quadrature rule (3.2.2), $Q = w_0 f_0 + w_1 f_1 + w_2 f_2$, Simpson's rule gives $w_0 = h/3$, $w_1 = 4h/3$, and $w_2 = h/3$ as weighting factors.

3.2.2 Composite rules

The integration methods (3.2.6d) give values for the integrals over only a small part of the integration domain. The trapezoid rule, for instance, only gives a value for $x \in [x_0, x_1]$. However, our fundamental aim is to evaluate the integral over the entire domain $x \in [a, b]$. Assuming once again that our interval is divided as $a = x_0 < x_1 < x_2 < \cdots < x_N = b$, the trapezoid rule applied over the interval gives the total integral

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=1}^{N-1} \frac{h}{2}(f_j + f_{j+1}). \quad (3.2.9)$$

Writing out this sum gives

$$\begin{aligned} \sum_{j=1}^{N-1} \frac{h}{2}(f_j + f_{j+1}) &= \frac{h}{2}(f_0 + f_1) + \frac{h}{2}(f_1 + f_2) + \cdots + \frac{h}{2}(f_{N-1} + f_N) \\ &= \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{N-1} + f_N) \\ &= \frac{h}{2}\left(f_0 + f_N + 2\sum_{j=1}^{N-1} f_j\right). \end{aligned} \quad (3.2.10)$$

The final expression no longer double counts the values of the points between f_0 and f_N. Instead, the final sum only counts the intermediate values once, thus making the algorithm about twice as fast as the previous sum expression. These are computational savings, which should always be exploited if possible.

3.2.3 Recursive improvement of accuracy

Given an integration procedure and a value of h, a function or data set can be integrated to a prescribed accuracy. However, it may be desirable to improve the accuracy without having

to disregard previous approximations to the integral. To see how this might work, consider the trapezoidal rule for a step-size of $2h$. Thus, the even data points are the only ones of interest, and we have the basic one-step integral

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{2h}{2}(f_0 + f_2) = h(f_0 + f_2). \tag{3.2.11}$$

The composite rule associated with this is then

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=0}^{N/2-1} h(f_{2j} + f_{2j+2}). \tag{3.2.12}$$

Writing out this sum gives

$$\begin{aligned}\sum_{j=0}^{N/2-1} h(f_{2j} + f_{2j+2}) &= h(f_0 + f_2) + h(f_2 + f_4) + \cdots + h(f_{N-2} + f_N) \\ &= h(f_0 + 2f_2 + 2f_4 + \cdots + 2f_{N-2} + f_N) \\ &= h\left(f_0 + f_N + 2\sum_{j=1}^{N/2-1} f_{2j}\right).\end{aligned} \tag{3.2.13}$$

Comparing the middle expressions in (3.2.10) and (3.2.13) gives a great deal of insight into how recursive schemes for improving accuracy work. Specifically, we note that the more accurate scheme with step-size h contains all the terms in the integral approximation using step-size $2h$. Quantifying this gives

$$Q_h = \frac{1}{2}Q_{2h} + h(f_1 + f_3 + \cdots + f_{N-1}), \tag{3.2.14}$$

where Q_h and Q_{2h} are the quadrature approximations to the integral with step-size h and $2h$, respectively. This then allows us to cut the value of h in half and improve accuracy without jettisoning the work required to approximate the solution with the accuracy given by a step-size of $2h$. This recursive procedure can be continued so as to give higher accuracy results. Further, this type of procedure holds for Simpson's rule as well as any of the integration schemes developed here. This recursive routine is used in Python's integration routines in order to generate results to a prescribed accuracy.

3.3 Implementation of Differentiation and Integration

This section focuses on the implementation of differentiation and integration methods. Since they form the backbone of calculus, accurate methods to approximate these calculations are essential. To begin, we consider a specific function to be differentiated, namely a hyperbolic secant. Part of the reason for considering this function is that the exact values of its first two

derivatives are known. This allows us to make a comparison of our approximate methods with the actual solution. Thus, we consider

$$u = \operatorname{sech}(x) \tag{3.3.1}$$

whose derivative and second derivative are

$$\frac{du}{dx} = -\operatorname{sech}(x)\tanh(x) \tag{3.3.2a}$$

$$\frac{d^2u}{dx^2} = \operatorname{sech}(x) - \operatorname{sech}^3(x). \tag{3.3.2b}$$

3.3.1 Differentiation

To begin the calculations, we define a spatial interval. For this example, we take the interval $x \in [-10, 10]$. In Python, the spatial discretization, Δx, must also be defined. This gives

```
dx = 0.1   # spatial discretization
x = np.arange(-10, 10 + dx, dx)  # spatial domain
```

Once the spatial domain has been defined, the function to be differentiated must be evaluated.

```
def sech(x):
    y=1/np.cosh(x)
    return y

def tanh(x):
    y=np.sinh(x)/np.cosh(x)
    return y

u = sech(x)
ux_exact = -sech(x) * tanh(x)
uxx_exact = sech(x) - sech(x)**3
```

Python *figure 1* produces the function and its first two derivatives.

To calculate the derivative numerically, we use the centre-, forward-, and backward-difference formulas derived for differentiation. Specifically, we will make use of the following four first-derivative approximations from Tables 3.1–3.3:

$$\text{centre-difference } O(h^2): \frac{y_{n+1} - y_{n-1}}{2h} \tag{3.3.3a}$$

$$\text{centre-difference } O(h^4): \frac{-y_{n+2} + 8y_{n+1} - 8y_{n-1} + y_{n-2}}{12h} \tag{3.3.3b}$$

$$\text{forward-difference } O(h^2): \frac{-3y_n + 4y_{n+1} - y_{n+2}}{2h} \tag{3.3.3c}$$

$$\text{backward-difference } O(h^2): \frac{3y_n - 4y_{n-1} + y_{n-2}}{2h}. \tag{3.3.3d}$$

Here we have used $h = \Delta x$ and $y_n = y(x_n)$.

3.3.2 Second-order accurate derivative

To calculate the second-order accurate derivative, we use the first, third, and fourth equations of (3.3.3d). In the interior of the domain, we use the centre-difference scheme. However, at the left and right boundaries, there are no left and right neighbouring points, respectively, to calculate the derivative with. Thus, we require the use of forward- and backward-difference schemes, respectively. This gives the basic algorithm

```
n = len(x)

# 2nd-order accurate finite difference
ux = np.zeros_like(x)
ux[0] = (-3 * u[0] + 4 * u[1] - u[2]) / (2 * dx)
ux[1:n-1] = (u[2:] - u[:n-2]) / (2 * dx)
ux[n-1] = (3 * u[n-1] - 4 * u[n-2] + u[n-3]) / (2 * dx)
```

The values of $ux(1)$ and $ux(n)$ are evaluated with the forward- and backward-difference schemes.

3.3.3 Fourth-order accurate derivative

A higher degree of accuracy can be achieved by using a fourth-order scheme. A fourth-order centre-difference scheme such as the second equation of (3.3.3d) relied on two neighbouring points. Thus, the first two and last two points of the computational domain must be handled separately. In what follows, we use a second-order scheme at the boundary points, and a fourth-order scheme for the interior. This gives the algorithm

```
# 4th-order accurate finite difference
ux2 = np.zeros_like(x)
ux2[:3] = ux[:3]  # Copy values from 2nd-order accurate scheme
ux2[2:n-2] = (-u[4:] + 8 * u[3:n-1] - 8 * u[1:n-3] + u[:n-4])
  / (12 * dx)
ux2[n-2:] = (3 * u[n-1] - 4 * u[n-2] + u[n-3]) / (2 * dx)
```

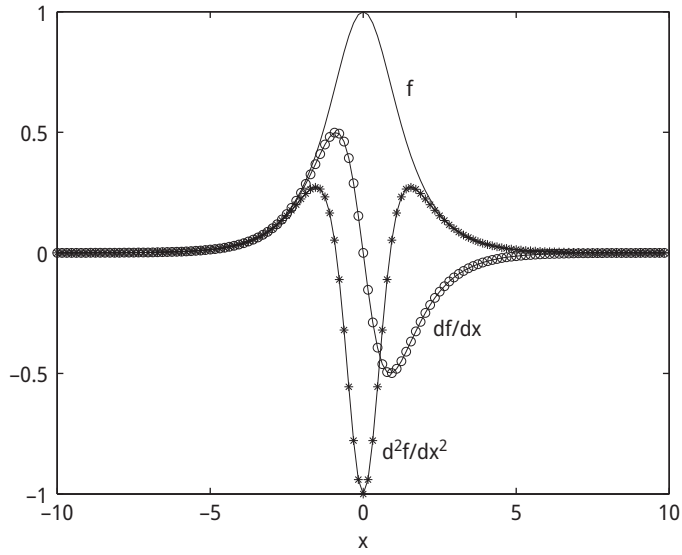


Figure 3.5 Calculation of the first and second derivative of $f(x) = \mathrm{sech}(x)$ using the finite-difference method. The approximate solutions for the first and second derivative are compared with the exact analytic solutions (circles and stars, respectively).

For the $\Delta x = 0.1$ considered here, the second-order and fourth-order schemes should result in errors of 10^{-2} and 10^{-4}, respectively.

To view the accuracy of the first derivative approximations, the results are plotted together with the analytic solution for the derivative

```
plt.plot(x, ux_exact, 'o', label='u_x_exact')
plt.plot(x, ux, 'c', label='u_x_2nd_order')
plt.plot(x, ux2, 'm', label='u_x_4th_order')
plt.legend()
```

To the naked eye, these results all look to be on the exact solution. However, by zooming in on a particular point, it quickly becomes clear that the errors for the two schemes are indeed 10^{-2} and 10^{-4}, respectively. The failure of accuracy can also be observed by taking large values of Δx. For instance, $\Delta x = 0.5$ and $\Delta x = 1$ illustrate how the derivatives fail to be properly determined with large Δx values. Figure 3.5 shows the function and its first two numerically calculated derivatives for the simple function $f(x) = \mathrm{sech}(x)$. A more careful comparison of the accuracy of both schemes is given in Fig. 3.6. This figure shows explicitly that for $\Delta x = 0.1$, the second-order and fourth-order schemes produce errors on the order of 10^{-2} and 10^{-4}, respectively.

As a final note, if you are given a set of data to differentiate, it is always recommended that you first run a spline through the data with an appropriately small Δx and then differentiate

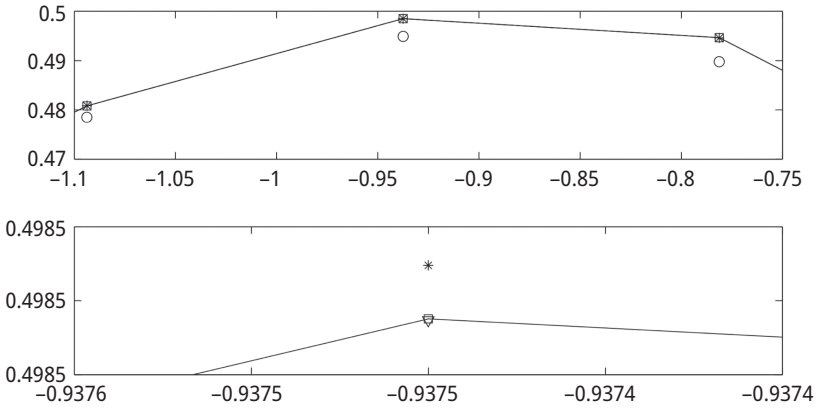


Figure 3.6 Accuracy comparison between second- and fourth-order finite difference methods for calculating the first derivative. Note that by using the *axis* command, the exact solution (line) and its approximations can be magnified from Fig. 3.5 near an arbitrary point of interest. Here, the top figure shows that the second-order finite difference (circles) method is within $O(10^{-2})$ of the exact derivative (box). The fourth-order finite difference (star) is within $O(10^{-5})$ of the exact derivative.

the spline. This will help to give smooth differentiated data. Otherwise, the data will tend to be highly inaccurate and choppy.

3.3.4 Integration

There are a large number of integration routines built into Python. So unlike the differentiation routines presented here, we will simply make use of the built-in Python functions. The most straightforward integration routine is the trapezoidal rule. Given a set of data, the **trapz** command can be used to implement this integration rule. For the x and u data defined previously for the hyperbolic secant, the command structure is

```
int_sech = trapz(u**2, x)
```

where we have integrated $\mathrm{sech}^2(x)$. The value of this integral is exactly 2. The **trapz** command gives a value that is within 10^{-7} of the true value. To generate the cumulative values, the **cumtrapz** command is used

```
int_sech2 = integrate.cumtrapz(u**2, x)
plt.plot(x[1:], int_sech2)
```

Python *figure 3* gives the value of the integral as a function of x. Thus, if the function u represented the velocity of an object, then the vector **int_sech2** generated from **cumtrapz** would represent the position of the object.

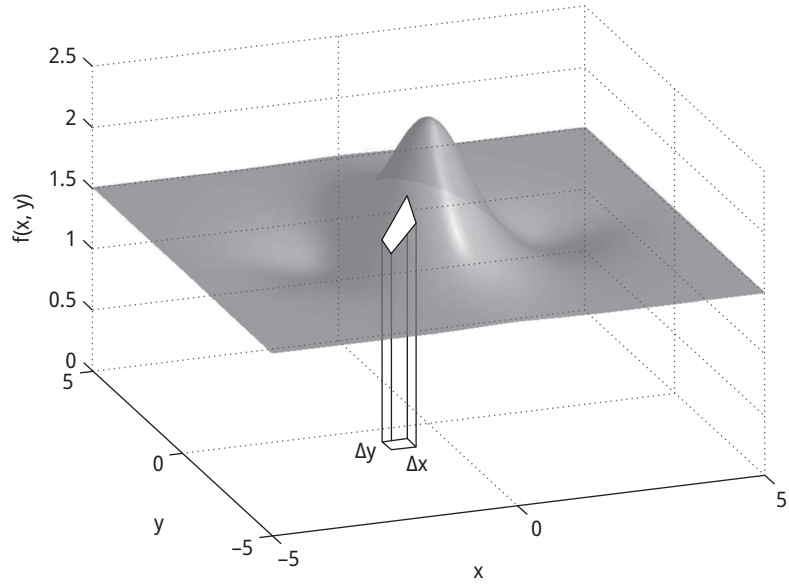


Figure 3.7 Two-dimensional trapezoidal rule integration procedure. As with one dimension, the domain is broken down into a grid of width and height Δx and Δy. The area of each parallelepiped is then easily calculated and summed up.

Alternatively, a function can be specified for integration over a given domain. The **quad** command is implemented by specifying a function and the range of integration. This will give a value of the integral using a recursive Simpson's rule that is accurate to within 10^{-6}. To integrate the function we have been considering thus far, the specific command structure is as follows:

```
A = 1
int_quad, _ = quad(lambda x: A * sech(x)**2, -10, 10)
```

Here the **inline** command allows us to circumvent a traditional function call. The **quad** command can, however, be used with a function call. This command executes the integration of $\mathrm{sech}^2(x)$ over $x \in [-10, 10]$ with the parameter A. Often in computations, it is desirable to pass in these parameters. Thus, this example is particularly helpful in illustrating the full use of the **quad** command. The two empty brackets are for tolerance and trace settings.

Double and triple integrals over two-dimensional rectangles and three-dimensional boxes can be performed with the **dblquad** and **triplequad** commands. Note that no version of the **quad** command exists that produces cumulative integration values. However, this can be easily handled in a **for** loop. Double integrals can also be performed with the **trapz** command using a **trapz** imbedded within another **trapz**. As an example of a two-dimensional integration, we consider the **dblquad** command for integration of the following

two-dimensional function

$$f(x,y) = \cos(x)\text{sech}(x)\text{sech}(y) + 1.5 \tag{3.3.4}$$

on the box $x \in [-5,5]$ and $y \in [-5,5]$. The **trapz** Python code for solving this problem is the following

```
area, _ = dblquad(lambda x, y: np.cos(x)*sech(x)*sech(y)+1.5,-5,5,-5,5)
```

This integrates the function over the box of interest. Figure 3.7 demonstrates the trapezoidal rule that can be used to handle the two-dimensional integration procedure. A similar procedure is carried out in three dimensions.

3.4 Differentiation of Noisy Data

Although finite differences are easy to understand and implement, they are remarkably bad for differentiating data with noise. Specifically, noisy data is deleterious for estimating the slopes required for derivative estimation. The differentiation of noisy data is often overlooked in many numerical methods books since the assumption is that one is dealing with numerically accurate data. However, producing good derivative estimates is incredibly challenging and a diverse number of algorithms have been proposed in order to handle this task. In general, one should never use finite differences directly for producing derivative estimates of noisy data. In what follows, two simple algorithms are proposed for estimating derivatives: spline smoothing and the Savitzky–Golay filtering. Both are simple methods, which can greatly improve performance in the critical task of estimating derivatives.

3.4.1 Spline smoothing

Since the mid 1970s, splines have provided an ideal framework for smoothing and refining data for differentiation [13, 14]. By construction, splines are required to be continuous in both the function and its first derivative. This is typically achieved by using cubic interpolation functions between a given set of data points $[y_0, y_1, \ldots, y_n]$. Specifically, the interpolation function that goes through the data is given by

$$S_k(t) = a_k + b_k(t-t_k) + c_k(t-t_k)^2 + d_k(t-t_k)^3 \qquad t \in [t_k, t_{k+1}] \tag{3.4.1}$$

for $k = 0, 2, \ldots, n-1$. The parameters of the cubic, a_k, b_k, c_k, and d_k, are determined by enforcing continuity of the function and its first derivative at the left and right boundary points of the range of validity of the spline

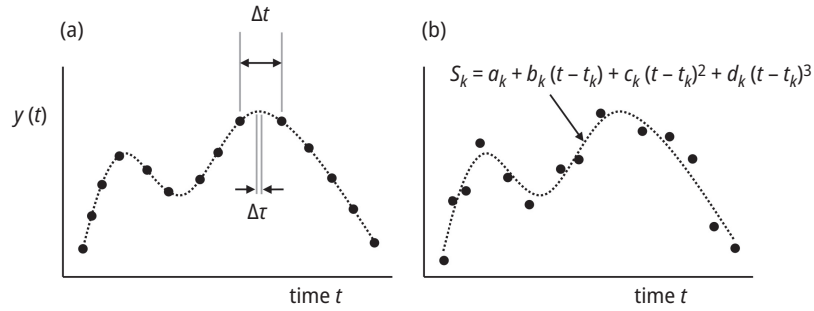


Figure 3.8 (a) Cubic spline fitting for refinement of a grid from Δt to $\Delta\tau$. The refinement can be used with finite differences to produce more accurate approximations to derivatives. (b) Cubic spline smoothing for handling noisy data. In this case, the spline is not forced to go through the data. Rather a balance between data fitting and smoothness is enforced by penalizing the second derivatives of the spline fit.

$$S_k(t_k) = S_{k-1}(t_k) = y_k \qquad \text{and} \qquad S_k(t_{k+1}) = S_{k+1}(t_{k+1}) = S_{k+1} \tag{3.4.2a}$$

$$\dot{S}_k(t_k) = \dot{S}_{k-1}(t_k) \qquad \text{and} \qquad \dot{S}_k(t_{k+1}) = \dot{S}_{k+1}(t_{k+1}). \tag{3.4.2b}$$

This gives four equations to uniquely determine the four parameters of the spline. At the left and right boundary points, y_0 and y_n, additional constraints must be imposed since (3.4.2b) no longer holds due to lack of neighbouring data points. A common constraint is to impose $\ddot{S}_0(t_0) = 0$ and $\ddot{S}_{n-1}(t_n) = 0$ on the second derivative in order to make the problem well posed.

For noiseless data, the spline construction allows one to refine coarsely spaced data, sampled at Δt, to a new refined sampling rate $\Delta\tau \ll \Delta t$ in order to produce improved derivative approximations with finite differences (see Fig. 3.8(a)). Alternatively, one can simply provide a computation of the derivative from the coefficients b_k, c_k, and d_k. Indeed, the splines are ideal as they are constructed by smoothness assumptions. However, for noisy data, enforcing (3.4.2b) is problematic as the spline fit now is forced to pass through the noisy data points, thus producing a poor representation of the underlying noiseless data. Specifically, the noisy data often produces stronger variations in the spline which is characterized by larger values of the derivatives. To circumvent this, cubic spline smoothing of the data can easily be modified so that it is not forced to pass through the data; thus, additional constraints are required for this scenario. To help mitigate the observed strong variations, a penalty on the value of the second derivative can be imposed. Thus, one would like to minimize the quantity $\sum_{j=0}^{n-1} \|\ddot{S}_k\|$ while ensuring that the spline stays close to the data. The smoothing spline can thus be found by optimization:

$$\mathbf{arg}_{a_k,b_k,c_k,d_k} \sum_{j=0}^{n-1} \lambda \left(\|S_k(t_k) - y_k\| + \|S_k(t_{k+1}) - y_{k+1}\| \right) + (1-\lambda) \left(\|\ddot{S}_k\| \right) \tag{3.4.3}$$

where the parameter λ is a hyper-parameter, which is tuned to balance fitting the data with keeping the second derivative small. In the limit $\lambda = 1$, the spline is forced to go through

each data point, while the limit $\lambda = 0$ penalizes the second derivative exclusively, resulting in a line fit to the data since this forces all $c_k = d_k = 0$. Figure 3.8(b) shows a representation of the cubic spline fitting procedure for noisy data and the resulting curve fit used for differentiation. The spline smoothing method works well if you need a smooth derivative with relatively few inflection points. It is fast but difficult to tune because the parameters can have dramatic effects. The two parameters to be tuned are: k, order of the spline, with larger values allowing for faster changes in the dynamics, and s, the smoothing factor, which is used to choose the number of knots. Splines work best with high-resolution data.

3.4.2 Savitzky–Golay Filtering

One of the most common methods for smoothing data before estimating the derivative is to apply local, polynomial least-square fitting to the data. The local aspect of the fitting makes this a convolutional technique. Abraham Savitzky and Marcel J. E. Golay [15] developed this method in the context of analytic chemistry data in the mid-1960s. Specifically, they showed that a moving polynomial fit can be numerically handled in exactly the same way as a weighted moving average, since the coefficients of the smoothing procedure are constant for all $\mathbf{y}$ values.

The goal of the Savitzky-Golay filter is to approximate the data $\mathbf{y}$ with a smoother version of the data $\mathbf{Y}$. This can then be used to estimate derivatives by, for instance, finite differences. Each position of the smoothed data is given by a weighted sum

$$Y_k = \sum_{j=(1-m)/2}^{(m-1)/2} c_j y_{k+j} \tag{3.4.4}$$

where $(m+1)/2 \geq k \geq n-(m-1)/2$, m is the number of points used for fittings, and $k = 1, 2, \ldots, n$ denotes all the data points. The m convolutional coefficients c_j are determined by least-square fitting of a given polynomial to the data. Least-square fitting to polynomials produces a linear system of equations for these coefficients. These coefficients, which are constant for all $\mathbf{y}$ values, can be explicitly computed for a polynomial of degree p, thus making the Savitzky-Golay filter easy to implement. As an example, consider approximating the function y_k using two neighbours on each side and a quadratic polynomial, as shown in Fig. 3.9. The approximate value of the smoothed function is then given by

$$Y_k = \frac{1}{35}\left(-3y_{k-2} + 12y_{k-1} + 17y_k + 12y_{k+1} - 3y_{k+2}\right) \tag{3.4.5}$$

so that $c_{\pm 2} = -3/35$ and $c_{\pm 1} = 12/35$. Such formulas are reminiscent of finite-difference approximations for the derivatives. Thus, once the polynomial order (p) and the number of neighbours (m) are specified ($m \geq (p+1)$), the appropriate Savitzky-Golay filter formula can be applied to produce a smoothed data set $\mathbf{Y}$, which approximates $\mathbf{y}$. The derivative can be approximated either directly from the polynomial of the fit or by then finite-differences, for instance, of the smoothed data.

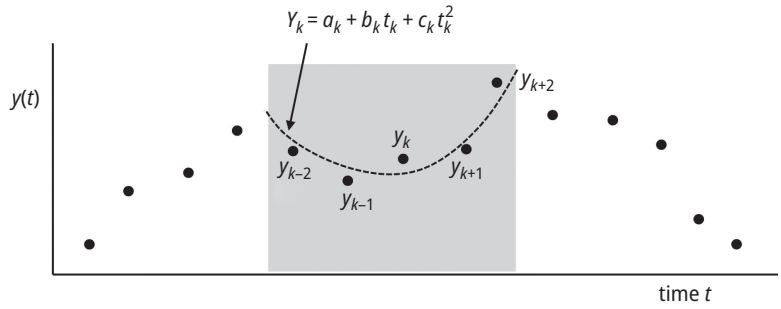


Figure 3.9 Local least-squares polynomial smoothing to the data point y_k. In this case, a quadratic fit $Y_k = a_k + b_k t_k + c_k t_k^2$ is used to approximate $y_k \approx Y_k$ using two neighbouring points on each side (shaded region). The Savitzky-Golay smoothing filter is like a weighted moving average since the coefficients of the smoothing procedure are constant for all $\mathbf{y}$ values.

3.4.3 Alternative Methods

Since the finite difference derivative dramatically amplifies the noise in a noisy time series. This amplification can be mitigated by adding some regularization (e.g. a penalty) to the calculation of the derivative. One approach is to penalize the total variation of the derivative. The total variation (TV) of a discrete time series is the sum of the absolute value of the differences between neighbouring values. This approach to calculating a smoother derivative was introduced in Osher and Fatemi [16] in the context of noisy images. This was later formulated as the total variation regularized derivative for noisy time-series [17]. Solving for the total variation regularized derivative involves calculating the finite difference derivative, subject to the following loss function,

$$L = (\mathbf{x} - \hat{\mathbf{x}})^2 + \gamma \, TV\left(\frac{d^n \hat{\mathbf{x}}}{dt^n}\right), \tag{3.4.6}$$

where larger values of γ result in smoother derivatives. If γ is zero, this derivative amounts to a (non-centred) finite difference derivative. The smoothed data can then be differentiated using the finite difference method to determine an approximation for the derivative.

If additional information about the dynamics of the system is available, such as a model of the dynamics, and/or other relevant synchronized data streams, then this additional knowledge can be incorporated into the estimates to calculate a more accurate derivative of the state of interest. A common framework for taking advantage of this additional information is the Kalman filter/smoother. With Kalman filters, smoothed estimates are calculated using a weighted sum of the noisy measurements and predicted values that are based on a dynamical systems model. The better the model, the more its prediction can be trusted, and the smoother the estimate can be without sacrificing accuracy. Furthermore, the model makes it possible to estimate states that are not explicitly measured, such as the derivative of any given state. Kalman estimators dominate the engineering sciences as a practical way to work with real data and problems [18].

3.4.4 PyNumDiff

PyNumDiff is a Python package that implements various methods for computing numerical derivatives of noisy data, which can be a critical step in developing dynamic models or designing control. There are four different families of methods implemented in this repository: smoothing followed by finite difference calculation, local approximation with linear models, Kalman filtering-based methods, and total variation regularization methods. Most of these methods have multiple parameters involved to tune. The package takes a principled approach and proposes a multi-objective optimization framework for choosing parameters that minimize a loss function to balance the faithfulness and smoothness of the derivative estimate. One can easily clone this repository in order to access some of the best numerical differentiation algorithms available today.

4 Curve Fitting

4.1 Least-Square Fitting Methods 90
4.1.1 Least-squares line 91
4.1.2 Data linearization 93
4.1.3 General fitting 94
4.2 Polynomial Fits and Splines 95
4.2.1 Splines 97
4.3 Data Fitting with Python 99
4.3.1 Least-squares fitting 99
4.3.2 Interpolation 101
4.3.3 Nonpolynomial least-square fitting 102
4.4 Sparsity for Learning a Curve Fit 104

Analysing data is fundamental to any aspect of science. Often data can be noisy in nature and only the trends in the data are sought. A variety of curve fitting schemes can be generated to provide simplified descriptions of data and its behaviour. The least-squares fit method is explored along with fitting methods of polynomial fits and splines.

4.1 Least-Square Fitting Methods

One of the fundamental tools for data analysis and recognizing trends in physical systems is curve fitting. The concept of curve fitting is fairly simple: use a simple function to describe a trend by minimizing the error between the selected function to fit and a set of data. The mathematical aspects of this are laid out in this section.

Suppose we are given a set of n data points

$$(x_1, y_1),\ (x_2, y_2),\ (x_3, y_3),\ \ldots\ , (x_n, y_n)\,. \tag{4.1.1}$$

Further, assume that we would like to fit a best fit line through these points. Then, we can approximate the line by the function

$$f(x) = Ax + B \tag{4.1.2}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0005

where the constants A and B are chosen to minimize some error. Thus, the function gives an approximation to the true value which is off by some error so that

$$f(x_k) = y_k + E_k \tag{4.1.3}$$

where y_k is the true value and E_k is the error from the true value.

Various error measurements can be minimized when approximating with a given function $f(x)$. Three standard possibilities are given as follows:

1. Maximum Error:

$$E_\infty(f) = \max_{1<k<n} |f(x_k) - y_k|. \tag{4.1.4a}$$

2. Average Error :

$$E_1(f) = \frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|. \tag{4.1.4b}$$

3. Root-mean Square :

$$E_2(f) = \left(\frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|^2\right)^{1/2}. \tag{4.1.4c}$$

In practice, the root-mean square error is most widely used and accepted. Thus, when fitting a curve to a set of data, the root-mean square error is chosen to be minimized. This is called a *least-square fit*. Figure 4.1 depicts three line fits for the errors E_∞, E_1, and E_2 listed above. The E_∞ error line fit is strongly influenced by the one data point that does not fit the trend. The E_1 and E_2 line fit nicely through the bulk of the data.

4.1.1 Least-squares line

To apply the least-square fit criteria, consider the data points $\{x_k, y_k\}$, where $k = 1, 2, 3, \ldots, n$. To fit the curve

$$f(x) = Ax + B \tag{4.1.5}$$

to this data, the E_2 is found by minimizing the sum

$$E_2(f) = \sum_{k=1}^{n} |f(x_k) - y_k|^2 = \sum_{k=1}^{n} (Ax_k + B - y_k)^2 . \tag{4.1.6}$$

Minimizing this sum requires differentiation. Specifically, the constants A and B are chosen so that a minimum occurs; thus we require: $\partial E_2/\partial A = 0$ and $\partial E_2/\partial B = 0$. Note that although a zero derivative can indicate either a minimum or maximum, we know this must

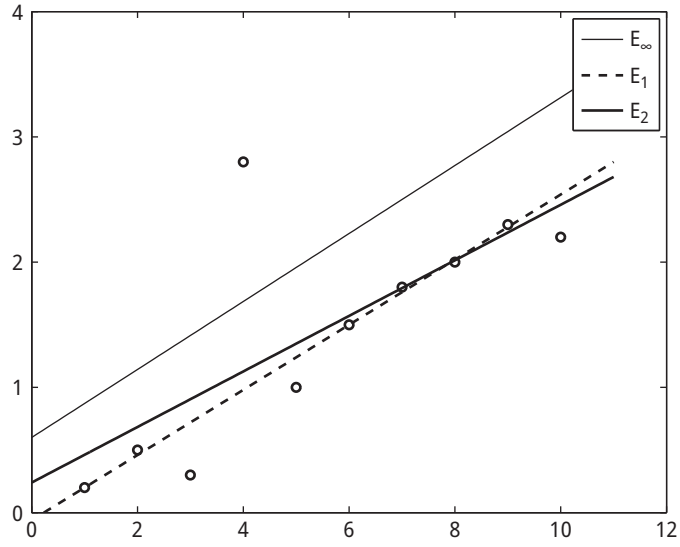


Figure 4.1 Linear fit $Ax + B$ to a set of data (open circles) for the three error definitions given by E_∞, E_1, and E_2. The weakness of the E_∞ fit results from the one stray data point at $x = 4$.

be a minimum to the error since there is no maximum error, i.e. we can always choose a line that has a larger error. The minimization condition gives:

$$\frac{\partial E_2}{\partial A} = 0: \quad \sum_{k=1}^{n} 2(Ax_k + B - y_k)x_k = 0 \tag{4.1.7a}$$

$$\frac{\partial E_2}{\partial B} = 0: \quad \sum_{k=1}^{n} 2(Ax_k + B - y_k) = 0\,. \tag{4.1.7b}$$

Upon rearranging, the 2×2 system of linear equations is found for A and B:

$$\begin{pmatrix} \sum_{k=1}^{n} x_k^2 & \sum_{k=1}^{n} x_k \\ \sum_{k=1}^{n} x_k & n \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} x_k y_k \\ \sum_{k=1}^{n} y_k \end{pmatrix}. \tag{4.1.8}$$

This equation can be easily solved using the backslash command in Python.

This method can be easily generalized to higher polynomial fits. In particular, a parabolic fit to a set of data requires the fitting function

$$f(x) = Ax^2 + Bx + C \tag{4.1.9}$$

where now the three constants A, B, and C must be found. These can be found from the 3×3 system, which results from minimizing the error $E_2(A, B, C)$ by taking

$$\frac{\partial E_2}{\partial A} = 0 \tag{4.1.10a}$$

$$\frac{\partial E_2}{\partial B} = 0 \tag{4.1.10b}$$

$$\frac{\partial E_2}{\partial C} = 0\,. \tag{4.1.10c}$$

4.1.2 Data linearization

Although a powerful method, the minimization procedure can result in equations, which are nontrivial to solve. Specifically, consider fitting data to the exponential function

$$f(x) = C\exp(Ax)\,. \tag{4.1.11}$$

The error to be minimized is

$$E_2(A,C) = \sum_{k=1}^{n}(C\exp(Ax_k) - y_k)^2\,. \tag{4.1.12}$$

Applying the minimizing conditions leads to

$$\frac{\partial E_2}{\partial A} = 0: \quad \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)Cx_k\exp(Ax_k) = 0 \tag{4.1.13a}$$

$$\frac{\partial E_2}{\partial C} = 0: \quad \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)\exp(Ax_k) = 0\,. \tag{4.1.13b}$$

This in turn leads to the 2×2 system

$$C\sum_{k=1}^{n} x_k\exp(2Ax_k) - \sum_{k=1}^{n} x_k y_k\exp(Ax_k) = 0 \tag{4.1.14a}$$

$$C\sum_{k=1}^{n}\exp(Ax_k) - \sum_{k=1}^{n} y_k\exp(Ax_k) = 0\,. \tag{4.1.14b}$$

This system of equations is nonlinear and cannot be solved in a straightforward fashion. Indeed, a solution may not even exist. Or many solutions may exist. Section 2 of Chapter 5 describes a possible iterative procedure for solving this nonlinear system of equations.

To avoid the difficulty of solving this nonlinear system, the exponential fit can be *linearized* by the transformation

$$Y = \ln(y) \tag{4.1.15a}$$

$$X = x \tag{4.1.15b}$$

$$B = \ln C\,. \tag{4.1.15c}$$

Then the fit function

$$f(x) = y = C\exp(Ax) \tag{4.1.16}$$

can be linearized by taking the natural log of both sides so that

$$\ln y = \ln(C\exp(Ax)) = \ln C + \ln(\exp(Ax)) = B + Ax \to Y = AX + B\,. \tag{4.1.17}$$

So by fitting to the natural log of the y-data

$$(x_i, y_i) \to (x_i, \ln y_i) = (X_i, Y_i) \tag{4.1.18}$$

the curve fit for the exponential function becomes a linear fitting problem that is easily handled.

4.1.3 General fitting

Given the preceding examples, a theory can be developed for a general fitting procedure. The key idea is to assume a form of the fitting function

$$f(x) = f(x, C_1, C_2, C_3, \ldots, C_M) \tag{4.1.19}$$

where the C_i are constants used to minimize the error and $M < n$. The root-mean square error is then

$$E_2(C_1, C_2, C_3, \ldots, C_m) = \sum_{k=1}^{n} (\, f(x_k, C_1, C_2, C_3, \ldots, C_M) - y_k)^2 \tag{4.1.20}$$

which can be minimized by considering the $M \times M$ system generated from

$$\frac{\partial E_2}{\partial C_j} = 0 \quad j = 1, 2, \ldots, M\,. \tag{4.1.21}$$

In general, this gives the *nonlinear* set of equations

$$\sum_{k=1}^{n} (\, f(x_k, C_1, C_2, C_3, \ldots, C_M) - y_k) \frac{\partial f}{\partial C_j} = 0 \quad j = 1, 2, 3, \ldots, M\,. \tag{4.1.22}$$

Solving this set of equations can be quite difficult. Most attempts at solving nonlinear systems are based upon iterative schemes, which require good initial guesses to converge to the solution. Regardless, the general fitting procedure is straightforward and allows for the construction of a best fit curve to match the data. Section 2 of Chapter 5 gives an example of a specific algorithm used to solve such nonlinear systems. In such a solution procedure, it is imperative that a reasonable initial guess be provided for by the user. Otherwise, the desired rapid convergence or convergence to the desired root may not be achieved.

4.2 **Polynomial Fits and Splines**

One of the primary reasons for generating data fits from polynomials, splines, or least-square methods is to *interpolate* or *extrapolate* data values. In practice, when considering only a finite number of data points

$$\begin{gathered}(x_0, y_0)\\(x_1, y_1)\\\vdots\\(x_n, y_n)\end{gathered}$$

the value of the curve at points other than the x_i are unknown. *Interpolation* uses the data points to predict values of $y(x)$ at locations where $x \neq x_i$ and $x \in [x_0, x_n]$. *Extrapolation* is similar, but it predicts values of $y(x)$ for $x > x_n$ or $x < x_0$, i.e. outside the range of the data points.

Interpolation and extrapolation are easy to do given a least-squares fit. Once the fitting curve is found, it can be evaluated for any value of x, thus giving an interpolated or extrapolated value. Polynomial fitting is another method for getting these values. With polynomial fitting, a polynomial is chosen to go through all data points. For the $n+1$ data points given above, an nth-degree polynomial is chosen

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{4.2.1}$$

where the coefficients a_j are chosen so that the polynomial passes through each data point. Thus, we have the resulting system

$$\begin{aligned}(x_0, y_0): \quad & y_0 = a_n x_0^n + a_{n-1}x_0^{n-1} + \cdots + a_1 x_0 + a_0\\(x_1, y_1): \quad & y_1 = a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0\\&\vdots\\(x_n, y_n): \quad & y_n = a_n x_n^n + a_{n-1}x_n^{n-1} + \cdots + a_1 x_n + a_0\,.\end{aligned}$$

This system of equations is nothing more than a linear system $\mathbf{Ax} = \mathbf{b}$, which can be solved by using the backslash command in Python. Note that unlike the least-square fitting method, the error of this fit is identically zero since the polynomial goes through each individual data point. However, this does not necessarily mean that the resulting polynomial is a good fit to the data. This will be shown in what follows.

Although this polynomial fit method will generate a curve that passes through all the data, it is an expensive computation since we have to first set up the system and then perform an $O(n^3)$ operation to generate the coefficients a_j. A more direct method for generating the relevant polynomial is the *Lagrange polynomials* method. Consider first the idea

of constructing a line between the two points (x_0, y_0) and (x_1, y_1). The general form for a line is $y = mx + b$, which gives

$$y = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0}\,. \tag{4.2.2}$$

Although valid, it is hard to continue to generalize this technique to fitting higher-order polynomials through a larger number of points. Lagrange developed a method, which expresses the line through the above two points as

$$p_1(x) = y_0\frac{x - x_1}{x_0 - x_1} + y_1\frac{x - x_0}{x_1 - x_0} \tag{4.2.3}$$

which can be easily verified to work. In a more compact and general way, this first-degree polynomial can be expressed as

$$p_1(x) = \sum_{k=0}^{1} y_k L_{1,k}(x) = y_0 L_{1,0}(x) + y_1 L_{1,1}(x) \tag{4.2.4}$$

where the *Lagrange coefficients* are given by

$$L_{1,0}(x) = \frac{x - x_1}{x_0 - x_1} \tag{4.2.5a}$$

$$L_{1,1}(x) = \frac{x - x_0}{x_1 - x_0}\,. \tag{4.2.5b}$$

Note the following properties of these coefficients:

$$L_{1,0}(x_0) = 1 \tag{4.2.6a}$$

$$L_{1,0}(x_1) = 0 \tag{4.2.6b}$$

$$L_{1,1}(x_0) = 0 \tag{4.2.6c}$$

$$L_{1,1}(x_0) = 1\,. \tag{4.2.6d}$$

By design, the Lagrange coefficients take on the binary values of zero or unity at the given data points. The power of this method is that it can be easily generalized to consider the $n + 1$ points of our original data set. In particular, we fit an nth-degree polynomial through the given data set of the form

$$p_n(x) = \sum_{k=0}^{n} y_k L_{n,k}(x) \tag{4.2.7}$$

where the Lagrange coefficient is

$$L_{n,k}(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}\,, \tag{4.2.8}$$

so that

$$L_{n,k}(x_j) = \begin{cases} 1 & j = k \\ 0 & j \neq k. \end{cases} \tag{4.2.9}$$

Thus, there is no need to solve a linear system to generate the desired polynomial. This is the preferred method for generating a polynomial fit to a given set of data and is the core algorithm employed by most commercial packages such as Python for polynomial fitting.

Figure 4.2 shows the polynomial fit that in theory has zero error. In this example, 15 data points were used and a comparison is made between the polynomial fit, least-square line fit, and a spline. The polynomial fit does an excellent job in the interior of the fitting regime but generates a phenomenon known as polynomial wiggle at the edges. This makes a polynomial fit highly suspect for any application due to its poor ability to accurately capture interpolations or extrapolations.

4.2.1 Splines

Although Python makes it a trivial matter to fit polynomials or least-square fits through data, a fundamental problem can arise as a result of a polynomial fit: *polynomial wiggle* (see Fig. 4.2). Polynomial wiggle is generated by the fact that an nth-degree polynomial has, in general, $n-1$ turning points from up to down or vice versa. One way to overcome this is to use a *piecewise polynomial interpolation* scheme. Essentially, this simply draws a line between

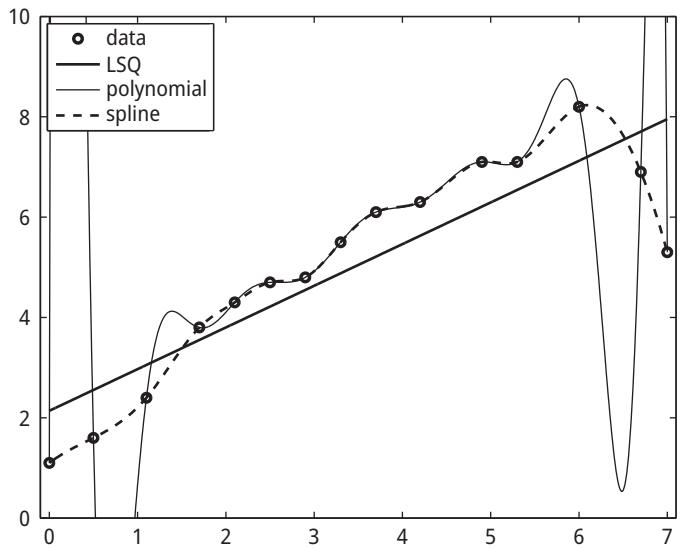


Figure 4.2 Fitting methods to a given data set (open circles) consisting of 15 points. Represented is a least-square (LSQ) line fit, a polynomial fit, and a spline. The polynomial fit has the drawback of generating polynomial wiggle at its edges making it a poor candidate for interpolation or extrapolation.

neighbouring data points and uses this line to give interpolated values. This technique is rather simple-minded, but it does alleviate the problem generated by polynomial wiggle. However, the interpolating function is now only a piecewise function. Therefore, when considering interpolating values between the points (x_0, y_0) and (x_n, y_n), there will be n linear functions each valid only between two neighbouring points.

The data generated by a piecewise linear fit can be rather crude and it tends to be choppy in appearance. *Splines* provide a better way to represent data by constructing cubic functions between points so that the first and second derivatives are continuous at the data points. This gives a smooth-looking function without polynomial wiggle problems. The basic assumption of the spline method is to construct a cubic function between data points:

$$S_k(x) = S_{k,0} + S_{k,1}(x - x_k) + S_{k,2}(x - x_k)^2 + S_{k,3}(x - x_k)^3 \tag{4.2.10}$$

where $x \in [x_k, x_{k+1}]$ and the coefficients $S_{k,j}$ are to be determined from various constraint conditions. Four constraint conditions are imposed:

$$S_k(x_k) = y_k \tag{4.2.11a}$$

$$S_k(x_{k+1}) = S_{k+1}(x_{k+1}) \tag{4.2.11b}$$

$$S'_k(x_{k+1}) = S'_{k+1}(x_{k+1}) \tag{4.2.11c}$$

$$S''_k(x_{k+1}) = S''_{k+1}(x_{k+1})\,. \tag{4.2.11d}$$

This allows for a smooth fit to the data since the four constraints correspond to fitting the data, continuity of the function, continuity of the first derivative, and continuity of the second derivative, respectively.

To solve for these quantities, a large system of equations $\mathbf{Ax} = \mathbf{b}$ is constructed. The number of equations and unknowns must first be calculated.

- $S_k(x_k) = y_k \rightarrow$ Solution fit: $n + 1$ equations;
- $S_k = S_{k+1} \rightarrow$ Continuity: $n - 1$ equations;
- $S'_k = S'_{k+1} \rightarrow$ Smoothness: $n - 1$ equations;
- $S''_k = S''_{k+1} \rightarrow$ Smoothness: $n - 1$ equations.

This gives a total of $4n - 2$ equations. For each of the n intervals, there are four parameters, which give a total of $4n$ unknowns. Thus, two extra constraint conditions must be placed on the system to achieve a solution. There are a large variety of options for assumptions, which can be made at the edges of the spline. It is usually a good idea to use the default unless the application involved requires a specific form. The spline problem is then reduced to a simple solution of an $\mathbf{Ax} = \mathbf{b}$ problem, which can be solved with the backslash command.

As a final remark on splines, splines are heavily used in computer graphics and animation. The primary reason is for their relative ease in calculating, and for their smoothness properties. There is an entire spline toolbox available for Python, which attests to the importance of

this technique for this application. Further, splines are also commonly used for smoothing data before differentiating. This will be considered further in upcoming chapters.

4.3 Data Fitting with Python

This section will discuss the practical implementation of the curve fitting schemes presented in the preceding two sections. The schemes to be explored are least-square fits, polynomial fits, line interpolation, and spline interpolation. Additionally, a nonpolynomial least-square fit will be considered, which results in a nonlinear system of equations. This nonlinear system requires additional insight into the problem and sophistication in its solution technique.

To begin the data fit process, we first import a relevant data set into the Python environment. To do so, the **loadtxt** command is used. The file *linefit.dat* is a collection of x and y data values put into a two-column format separated by spaces. The data is shown in Fig. 4.3. This data set is just an example of what you may want to import into Python. The command structure to read this data is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit, fmin
from scipy.interpolate import interp1d, splev, splrep

linefit = np.loadtxt('linefit.dat')
x = linefit[:, 0]
y = linefit[:, 1]
plt.plot(x, y, 'o:')
```

After reading in the data, the two vectors $\mathbf{x}$ and $\mathbf{y}$ are created from the first and second column of the data, respectively. It is this set of data, which will be explored with line fitting techniques. The code will also generate a plot of the data in *figure 1* in Python. Note that this is the data considered in Fig. 4.2 of the preceding section.

4.3.1 Least-squares fitting

The least-squares fit technique is considered first. The **polyfit** and **polyval** commands are essential to this method. Specifically, the **polyfit** command is used to generate the $n+1$ coefficients a_j of the nth-degree polynomial

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \tag{4.3.1}$$

used for fitting the data. The basic structure requires that the vectors $\mathbf{x}$ and $\mathbf{y}$ be submitted to **polyval** along with the desired degree of polynomial fit n. To fit a line ($n = 1$) through the data, use the command

```
pcoeff = np.polyfit(x, y, 1)
```

The output of this function call is a vector **pcoeff**, which includes the coefficients a_1 and a_0 of the line fit $p_1(x) = a_1x + a_0$. To evaluate and plot this line, values of x must be chosen. For this example, the line will be plotted for $x \in [0, 7]$ in steps of $\Delta x = 0.1$.

```
xp = np.arange(0, 7.1, 0.1)
yp = np.polyval(pcoeff, xp)
plt.plot(x, y, 'o', xp, yp, 'm')
```

The **polyval** command uses the coefficients generated from **polyfit** to generate the y-values of the polynomial fit at the desired values of x given by $\mathbf{xp}$. *Figure 2* in Python depicts both the data and the best line fit in the least-square sense. Figure 4.2 of the last section demonstrates a least-squares line fit through the data.

To fit a parabolic profile through the data, a second-degree polynomial is used. This is generated with

```
pcoeff2 = np.polyfit(x, y, 2)
yp2 = np.polyval(pcoeff2, xp)
plt.plot(x, y, 'o', xp, yp2, 'm')
```

Here the vector $\mathbf{yp2}$ contains the parabolic fit to the data evaluated at the x-values $\mathbf{xp}$. These results are plotted in Python *figure 3*. Figure 4.3 shows the least-square parabolic fit to the data. This can be contrasted to the linear least-square fit in Fig. 4.2. To find the least-square error, the sum of the squares of the differences between the parabolic fit and the actual data must be evaluated. Specifically, the quantity

$$E_2(f) = \left(\frac{1}{n} \sum_{k=1}^{n} |f(x_k) - y_k|^2 \right)^{1/2} \tag{4.3.2}$$

is calculated. For the parabolic fit considered in the last example, the polynomial fit must be evaluated at the x-values for the given data **linefit.dat**. The error is then calculated

```
yp3 = np.polyval(pcoeff2, x)
n = len(yp3)
E2 = np.sqrt(np.sum(np.abs(yp3 - y)**2) / n)
```

This is a quick and easy calculation, which allows for an evaluation of the fitting procedure. In general, the error will continue to drop as the degree of the polynomial is increased. This

is because every extra degree of freedom allows for a better least-squares fit to the data. Indeed, a degree $n-1$ polynomial has zero error since it goes through all the data points.

4.3.2 Interpolation

In addition to least-square fitting, interpolation techniques can be developed which go through all the given data points. The error in this case is zero, but each interpolation scheme must be evaluated for its accurate representation of the data. The first interpolation scheme is a polynomial fit to the data. Given $n+1$ points, an nth-degree polynomial is chosen. The **polyfit** command is again used for the calculation

```
n = len(x) - 1
pcoeffn = np.polyfit(x, y, n)
ypn = np.polyval(pcoeffn, xp)
plt.plot(x, y, 'o', xp, ypn, 'm')
```

The Python script will produce an nth-degree polynomial through the data points. But as always, there is the danger with polynomial interpolation: *polynomial wiggle* can dominate the behaviour. This is indeed the case as illustrated in *figure 4*. The strong oscillatory phenomenon at the edges is a common feature of this type of interpolation. Figure 4.2 has already illustrated the pernicious behaviour of the polynomial wiggle phenomenon. Indeed, the idea of using a high-degree polynomial for a data fit becomes highly suspect upon seeing this example.

In contrast to a polynomial fit, a piecewise linear fit gives a simple-minded connect-the-dot interpolation to the data. The **interp1** command gives the piecewise linear fit algorithm

```
yint = interp1d(x, y)(xp)
```

The linear interpolation is illustrated in *figure 5* in Python. There are a few options available with the **interp1** command, including the *nearest* option and the *spline* option. The *nearest* option gives the nearest value of the data to the interpolated value, while the *spline* option gives a cubic spline fit interpolation to the data. The two options can be compared with the default *linear* option.

```
yint = interp1d(x, y, kind='linear')(xp)
yint2 = interp1d(x, y, kind='nearest')(xp)
yint3 = interp1d(x, y, kind='cubic')(xp)
plt.plot(x, y, 'o', xp, yint, 'm', xp, yint2, 'k', xp, yint3, 'r')
```

Note that the spline option is equivalent to using the **spline** algorithm supplied by Python. Thus, a smooth fit can be achieved with either **spline** or **interp1**. Figure 4.3 demonstrates the

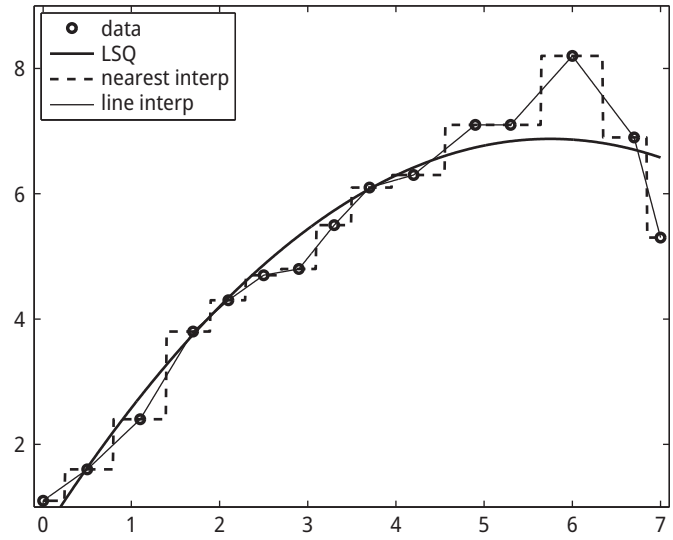


Figure 4.3 Fitting methods to a given data set (open circles) consisting of 15 points. Represented is a least-square (LSQ) quadratic fit, a nearest neighbour interpolation fit, and a linear interpolation between data points.

nearest neighbour and linear interpolation fit schemes. The most common (default) scheme is the simple linear interpolation between data points.

The **spline** command is used by giving the x and y data along with a vector $\mathbf{xp}$ for which we desire to generate corresponding y-values.

```
yspline = splev(xp, splrep(x, y))
plt.plot(x, y, 'o', xp, yspline, 'k')
```

The generated spline is depicted in Python *figure 6*. This is the same as that using **interp1** with the *spline* option. Note that the data is smooth as expected from the enforcement of continuous smooth derivatives. The spline fit was already demonstrated in the last section in Fig. 4.2.

4.3.3 Nonpolynomial least-square fitting

To consider more sophisticated least-square fitting routines, consider a data set, which looks like it could be nicely fitted with a Gaussian profile as shown in Fig. 4.4. To fit the data to a Gaussian, we indeed assume a Gaussian function of the form

$$f(x) = A\exp(-Bx^2)\,. \tag{4.3.3}$$

Following the procedure to minimize the least-square error leads to a set of nonlinear equations for the coefficients A and B. In general, solving a nonlinear set of equations can

be a difficult task. A solution is not guaranteed to exist. And in fact, there may be many solutions to the problem. Thus, the nonlinear problem should be handled with care.

To generate a solution, the least-square error must be minimized. In particular, the sum

$$E_2 = \sum_{k=0}^{n} |\, f(x_k) - y_k|^2 \tag{4.3.4}$$

must be minimized. The command **fminsearch** in Python minimizes a function of several variables. In this case, we minimize (4.3.4) with respect to the variables A and B. Thus, the minimum of

$$E_2 = \sum_{k=0}^{n} |A \exp(-Bx_k^2) - y_k|^2 \tag{4.3.5}$$

must be found with respect to A and B. The **fminsearch** algorithm requires a function call to the quantity to be minimized along with an initial guess for the parameters which are being used for minimization, i.e. A and B.

```
def gauss_fit(x0):
    # Load data
    gaussfit = np.loadtxt('gaussfit.dat')
    x = gaussfit[:, 0]
    y = gaussfit[:, 1]

    # Calculate error
    E = np.sum((x0[0] * np.exp(-x0[1] * x**2) - y)**2)

    return E


coeff = fmin(gauss_fit, [1, 1])
```

This command structure uses as initial guesses $A = 1$ and $B = 1$ when calling the file **gauss_fit.m**. After minimizing, it returns the vector $\mathbf{coeff}$ that contains the appropriate values of A and B, which minimize the least-square error.

The vector $\mathbf{x_0}$ accepts the initial guesses for A and B and is updated as A and B are modified through an iteration procedure to converge to the least-square values. Note that the sum is simply a statement of the quantity to be minimized, namely (4.3.5). The results of this calculation can be illustrated with Python *figure 7*:

```
xga = np.arange(-3, 3.1, 0.1)
a, b = coeff[0], coeff[1]
yga = a * np.exp(-b * xga**2)
plt.plot(xga, yga, 'm')
```

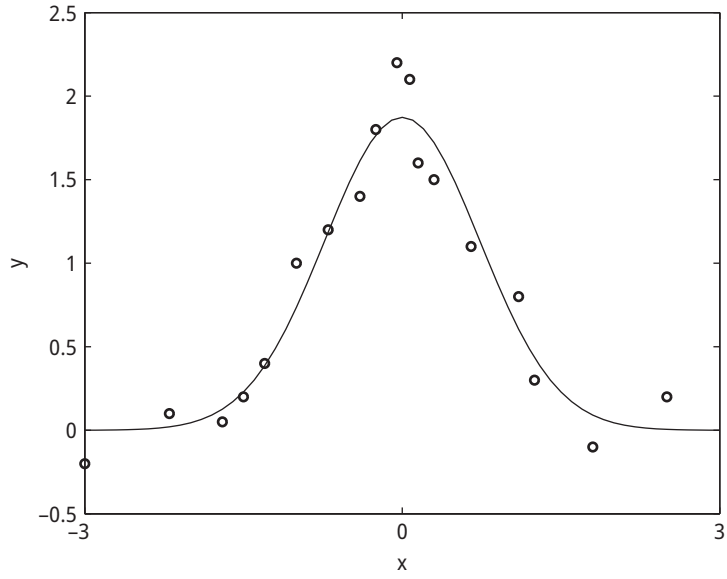


Figure 4.4 Gaussian fit $f(x) = A\exp(-Bx^2)$ to a given data set (open circles). The **fminsearch** command performs an iterative search to minimize the associated, nonlinear least-squares fit problem. It determines the values to be $A = 1.8733$ and $B = 0.9344$.

Note that for this case, the initial guess is extremely important. For any given problem where this technique is used, an educated guess for the values of parameters like A and B can determine if the technique will work at all. The results should also be checked carefully since there is no guarantee that a minimization near the desired fit can be found. Figure 4.4 shows the results of the least-square fit for the Gaussian example considered here. In this case, Python determines that $A = 1.8733$ and $B = 0.9344$ minimizes the least-square error.

4.4 Sparsity for Learning a Curve Fit

It is typical in curve fitting that the form of the curve is imposed by the user *a priori*. Thus, the form of the curve $f(x)$ is specified often by inspection of the data. From lines to parabolas to more exotic functional forms, least-square fitting is then computed through optimization. The form of $f(x)$ lends to a minimally parametrized and interpretable model of the data.

However, it can be the case that instead of specifying the curve to fit $f(x)$, one would like to directly learn a good curve fitting model. This is a data-driven approach to curve fitting that allows for a more agnostic approach to the curve fitting process. To formulate this problem, the key idea is to assume a form of the fitting function

$$f(x) = f(x, \xi_1, \xi_2, \xi_3, \ldots, \xi_M) \tag{4.4.1}$$

where the ξ_i are constants used to select the terms from a library of potential candidate fit functions. Specifically, the curve fitting procedure is now given as the following mathematical problem

$$\mathbf{y} = \boldsymbol{\Theta}\boldsymbol{\xi} \tag{4.4.2}$$

where

$$\boldsymbol{\Theta} = \begin{bmatrix} | & | & & | \\ \theta_1(\mathbf{x}) & \theta_1(\mathbf{x}) & \cdots & \theta_M(\mathbf{x}) \\ | & | & & | \end{bmatrix} \tag{4.4.3}$$

and the $\theta_k(\mathbf{x})$ are candidate functions for the fitting procedure. The vector

$$\boldsymbol{\xi} = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_M \end{bmatrix} \tag{4.4.4}$$

gives the weighting, or loading coefficient, for each of the candidate functions. To learn a fitting, the goal is to select a small subset of the $\theta_k(\mathbf{x})$, which best represent the data via a linear combination. Thus, the optimization procedure solves the linear system of equations (4.4.2) by promoting sparsity of the solution vector $\boldsymbol{\xi}$, i.e. the vector should be mostly zeros. This can be done by optimization where the sparsity promotion is imposed with the one-norm error E_1 defined by (4.1.4b). This explicitly gives

$$\min \|\boldsymbol{\xi}\|_1 \tag{4.4.5}$$

subject to (4.4.2). Such a method has been used in data decomposition methods [19] as well as for discovery of dynamic models from data [20, 21]. One of the most well-known algorithms for solving the sparsity promoting problem for $\mathbf{Ax} = \mathbf{b}$ is the LASSO [22].

To illustrate how to learn a model for the data via curve fitting, consider the following data generated from the two-dimensional function

$$F(x, y) = x^2 - xy^2 + \mathcal{N}(0, \sigma) \tag{4.4.6}$$

where $\mathcal{N}(0, \sigma)$ is noise that is added to the function, which has strength σ. The data is shown in the top left panel of Fig. 4.5.

The goal is to discover a good fit to the observed data by learning a representation of the function $f(x, y)$ through the sparse regression framework with a library of potential candidates $\boldsymbol{\Theta}$. For this example, the library of potential candidates is composed of polynomial functions of x and y which go up to cubics. This gives

$$\theta_k(x, y) \in \{1, x, y, x^2, xy, y^2, x^3, x^2y, xy^2, y^3\}. \tag{4.4.7}$$

This gives a total of ten candidate functions for fitting to the data. The following code generates the candidate functions and constructs the library matrix $\boldsymbol{\Theta}$.

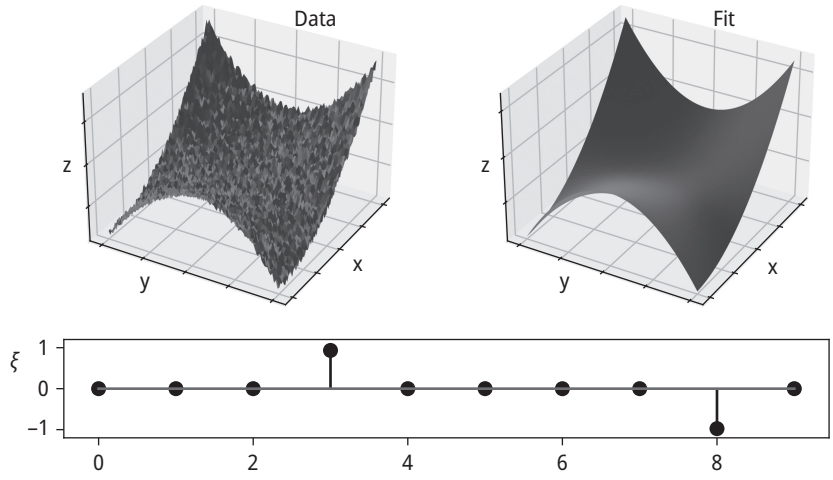


Figure 4.5 Sparse regression for data fitting. The noisy data is given in the top left panel. A library of potential candidate fitting functions, in this case given by polynomials which go up to cubics, is used to sparsely regress and find a functional form of the data. The discovered model is $\tilde{F}(x, y) = 0.94x^2 - 0.98xy^2$ with the loading coefficients $\boldsymbol{\xi}$ shown in the bottom panel. Only two terms from the library are shown to contribute.

```
nx = 100; ny = 100; N = nx*ny
x = np.linspace(-2, 2, nx)
y = np.linspace(-2, 2, ny)
x, y = np.meshgrid(x, y)

x2 = x.reshape(N,1)
y2 = y.reshape(N,1)
F = z.reshape(N,1)

th1 = 1 + 0*x2  # constant
th2 = x2 # linear terms
th3 = y2
th4 = x2*x2 # quadratic terms
th5 = x2*y2
th6 = y2*y2
th7 = x2*x2*x2 # cubic terms
th8 = x2*x2*y2
th9 = x2*y2*y2
th10 = y2*y2*y2

Theta = np.column_stack((th1, th2, th3, th4, th5, th6, th7,
        th8, th9, th10))
```

Note that the two-dimensional functions if reshaped into a single vector. Although tailored to dynamical systems, the pySINDy [23, 24] package provides a more comprehensive software architecture for constructing the library $\boldsymbol{\Theta}$.

The sparse regression is computed with the LASSO algorithm. Thus, once the library is constructed, it is fit to the data given by the function $F(x, y)$. This can be done with the following code

```
from sklearn.linear_model import Lasso

alpha = 0.1  # Regularization strength
lasso = Lasso(alpha=alpha)
lasso.fit(Theta, F)
xi = lasso.coef_
```

where the vector $\boldsymbol{\xi}$ is a sparse vector whose non-zero elements are of primary interest. The fitting procedure gives

$$\tilde{F}(x, y) = 0.94x^2 - 0.98xy^2 \tag{4.4.8}$$

with all other terms in the library of candidates set to zero, i.e. $\xi_0 = \xi_1 = \xi_2 = \xi_4 = \xi_5 = \xi_6 = \xi_7 = \xi_9 = 0$. Figure 4.5 shows both the model fit (top right panel) and the vector $\boldsymbol{\xi}$ values found from the LASSO regression.

5 Basic Optimization

5.1 Unconstrained Optimization (Derivative-Free Methods) 109
5.1.1 Golden section search 110
5.1.2 Successive parabolic interpolation 112
5.1.3 fminbnd 114
5.2 Unconstrained Optimization (Derivative Methods) 114
5.2.1 minimize 118
5.3 Linear Programming 119
5.3.1 Slack variables 120
5.3.2 A graphical interpretation 121
5.3.3 linprog 123
5.3.4 Open source optimization packages: cvx 124
5.4 Simplex Method 124
5.5 Genetic Algorithms 128
5.5.1 ga 132

Optimization methods are prevalent throughout a vast range of applications. In its simplest form, an optimization routine simply attempts to find the maximum or minimum of a real-valued function, i.e. the *objective function*, by developing an algorithm that systematically chooses input values from an allowed set, or *feasible* set, and computes the value of the function. Typically, the optimization algorithm is built upon an iteration scheme that continues to choose new input values so that the maximum or minimum of the objective function is achieved. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, optimization includes finding *best available* values of some objective function given a defined domain, including a variety of different types of objective functions and different types of domains. In the following sections, a brief introduction to these techniques will be given along with their Python function calls.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0006

5.1 Unconstrained Optimization (Derivative-Free Methods)

The concept of optimization is fairly simple: find the minimum or maximum of a function. In optimization, the function is real-valued and called the *objective function*. Moreover, it is often the case that you would like to find the maximum or minimum under some *constraints* on the input values to the function. Thus, there is a concept of *feasibility* of the solution. To begin, we will consider optimization without constraints, thus the aptly named *unconstrained optimization* problem. With a background in calculus, the mention of minimization or maximization of a function automatically evokes the concept of the derivative, i.e. setting the derivative of a function to zero implies it is either a minimum or maximum. In this section, however, derivative-free methods for computing the minimum or maximum will be considered. Such methods are often employed when computing the derivatives is either very expensive computationally, or simply intractable in practice.

Before proceeding forward with the development of the methods, abstracting the unconstrained optimization problem is necessary. Thus, consider minimizing the objective function

$$\min f(\mathbf{x}) \tag{5.1.1}$$

where the objective function depends generally on a number of variables

$$x_1, x_2, \ldots, x_n\,. \tag{5.1.2}$$

These are called the *control variables* because we can choose their values. Indeed, our objective is to choose these values so that the objective function (5.1.1) is minimized. It is also these control variables that are often constrained in practice so that it becomes a *constrained optimization* problem.

By definition, $f(\mathbf{x})$ has a minimum at a point $\mathbf{x} = \mathbf{x_0}$ in a region R if

$$f(\mathbf{x}) \geq f(\mathbf{x_0}) \quad \text{for } \mathbf{x} \in R. \tag{5.1.3}$$

A maximum can be defined in a similar way. However, a maximum of $f(\mathbf{x})$ becomes a minimum of $-f(\mathbf{x})$. Thus, it suffices from here forward to only consider the minimization problem. Note that in this definition of the minimum, it is not specified whether this is a local or global minimum. For highly nontrivial functions $f(\mathbf{x})$, it may be that there exists many local minima, thus a search algorithm that converges to a minimum solution may not be the actual desired solution. This will be illustrated shortly in practice.

For simplicity, we will consider derivative-free optimization methods for a function of a single variable ($\min f(x)$). The methods here are very much like the root solving techniques demonstrated in the earlier chapters of the book, i.e. they are simple iterative schemes that progressively produce better and better minimization values. In both methods shown, an initial interval is required in which the minimum lies.

5.1.1 Golden section search

Given an interval in which the minimum is known to exist, the golden section search algorithm is an efficient method for finding the minimum of the function. To be more specific, the function must be *unimodal* in the interval $x \in [a, b]$, meaning there is only a single relative minimum in the interval chosen. The algorithm is much like the bisection method for root solving. Specifically, within the interval where the minimum exists, two points (x_1 and x_2) are chosen so that $a < x_1 < x_2 < b$. Evaluation of the function $f(x)$ is then performed at x_1 and x_2. The following observations hold:

$$\text{if } f(x_1) \leq f(x_2) \quad \text{then retain interval } x \in [a, x_2]$$
$$\text{if } f(x_1) > f(x_2) \quad \text{then retain interval } x \in [x_1, b].$$

This ensures that the minimum is now in a smaller subinterval of the original $x \in [a, b]$.

The golden section search gives a simple procedure for selecting the evaluation points x_1 and x_2. Specifically, they are chosen so that (i) we want a constant reduction factor, say c, for the size of the interval, and (ii) they can be reused at the next iteration, thus saving a function evaluation. From the observations above, criterion (i) is enforced mathematically with the conditions:

$$c = \frac{x_2 - a}{b - a} \;\rightarrow\; x_2 = (1 - c)a + cb \tag{5.1.4a}$$
$$c = \frac{b - x_1}{b - a} \;\rightarrow\; x_1 = ca + (1 - c)b \tag{5.1.4b}$$

where c is the reduction factor to the new intervals $x \in [a, x_2]$ and $x \in [x_1, b]$, respectively (see Fig. 5.1). The key observation is the following: if $f(x_1) < f(x_2)$, then the new interval is $x \in [a, x_2]$. Moreover, a new x_1 must be evaluated while the old x_2 is the x_1. Thus, we have the following upon making use of (5.1.4b):

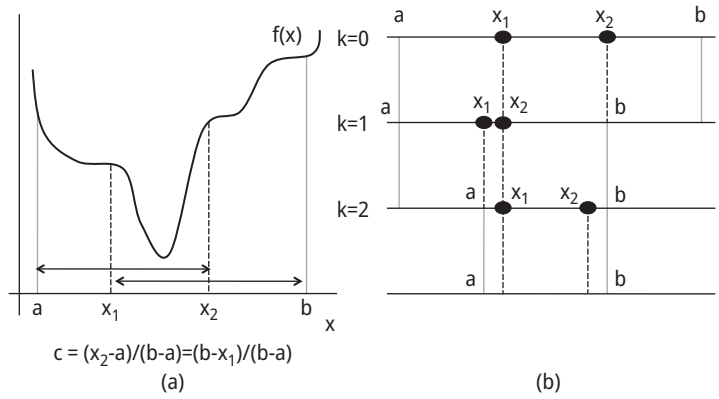


Figure 5.1 Graphical depiction of the golden section search. Two points, x_1 and x_2, are used for the iteration process. The points are chosen so as to keep the ratio of $(b - x_1)/(b - a)$ and $(x_2 - a)/(b - a)$ the same. The first three iterations are depicted.

$$x_2^{\text{new}} = x_1^{\text{old}} = ca + (1-c)b \tag{5.1.5a}$$

$$x_2^{\text{new}} = (1-c)a + cx_2^{\text{old}} = (1-c)a + c[(1-c)a + cb]. \tag{5.1.5b}$$

But these two new values of x_2 must be equivalent, thus we have

$$\begin{aligned} ca + (1-c)b &= (1-c)a + c[(1-c)a + cb] \\ c^2(b-a) + c(b-a) - (b-a) &= 0 \\ c^2 + c - 1 &= 0 \\ c &= (-1+\sqrt{5})/2 \approx 0.6180 \end{aligned} \tag{5.1.6}$$

where only the positive root of c is kept since c must be positive. Note that the resulting ratio is simply unity minus the golden ratio, i.e. $\phi = 1 + c$, thus the name *golden section search*. Now that c is determined, x_1 and x_2 can be found from (5.1.4b). Such a routine can be easily implemented in Python. The following finds the minimum of the function $f(x) = x^4 + 10x\sin(x^2)$ on the interval $x \in [-2, 1]$.

```
import numpy as np
a = -2; b = 1 # Initial interval
c = (-1 + np.sqrt(5)) / 2  # Golden section

x1 = c * a + (1 - c) * b
x2 = (1 - c) * a + c * b
f1 = x1**4 + 10 * x1 * np.sin(x1**2)
f2 = x2**4 + 10 * x2 * np.sin(x2**2)

for j in range(100):
    if f1 < f2:    # Move right boundary
        b = x2; x2 = x1; f2 = f1
        x1 = c * a + (1 - c) * b
        f1 = x1**4 + 10 * x1 * np.sin(x1**2)
else:         # Move left boundary
   a = x1; x1 = x2; f1 = f2
   x2 = (1 - c) * a + c * b
   f2 = x2**4 + 10 * x2 * np.sin(x2**2)

if (b - a) < 10**(-6):   # Break if close
   break
```

The above algorithm converges in 31 iterations to the minimum $f = -10.0882$ at $x = -1.2742$ with an accuracy of 10^{-6}. A theorem regarding the golden section search algorithm states that after k iterations, starting from the interval $x \in [a, b]$, the midpoint of this final interval is within $c^k(b-a)/2$ of the minimum. Thus, a guaranteed convergence rate can be established.

5.1.2 Successive parabolic interpolation

In the golden section search, no information was used about the values of $f(x_1)$ and $f(x_2)$ in selecting a new subinterval. Thus, if $f(x_1) \ll f(x_2)$, it would be judicious to assume that the minimum might be closer to the point x_1 than x_2 and the interval should cut accordingly. A technique that makes use of the function evaluation in choosing how to refine the interval is the method of successive parabolic interpolation.

The idea behind successive parabolic interpolation is to choose three points near the vicinity of the minimum: x_1, x_2, and x_3. These points correspond to left, middle, and right points, respectively (see Fig. 5.2). With the choice of these three points, one can evaluate the function values associated with each point, $f(x_1), f(x_2)$, and $f(x_3)$ and fit a parabola through them. Using the Lagrange polynomial coefficients, this gives a parabolic function

$$p(x) = f(x_1)\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + f(x_2)\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + f(x_3)\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \quad (5.1.7)$$

where $p(x)$ is now locally approximating the function $f(x)$. The minimum of the parabola now serves as a temporary proxy for the minimum of the actual function $f(x)$. The minimum of the parabola can be found by setting the first derivative to zero so that we evaluate $p'(x_0) = 0$. This gives, after some algebra, the following minimum:

$$x_0 = \frac{x_1 + x_2}{2} - \frac{(f_2 - f_1)(x_3 - x_1)(x_3 - x_2)}{2[(x_2 - x_1)(f_3 - f_2) - (f_2 - f_1)(x_3 - x_2)]}. \quad (5.1.8)$$

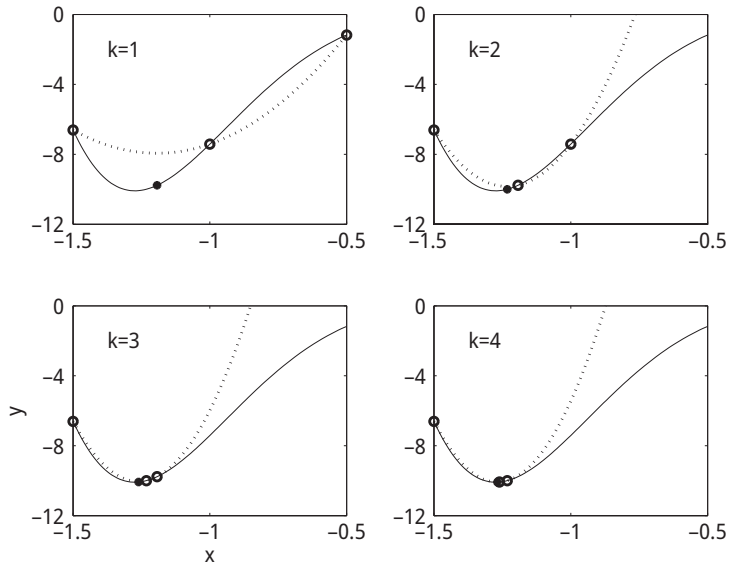


Figure 5.2 Graphical depiction of the successive parabolic interpolation algorithm. The three data points are depicted (circles) along with the point evaluated at the minimum of the parabola (star). The solid line is the function and the dotted line is the parabola generated. In this case, the guesses give a rapid convergence (14 iterations for 10^{-6} accuracy) to the minimum. However, the method is not guaranteed to converge and can, in fact, easily diverge. Thus it is important to have a good local starting point.

The idea now is to use this new point x_0 as our new middle point x_2. There are two cases of interest:

$$
\begin{aligned}
x_0 < x_2: \quad & x_1^{\text{new}} = x_1^{\text{old}} \\
& x_2^{\text{new}} = x_0 \\
& x_3^{\text{new}} = x_2^{\text{old}} \\
x_0 > x_2: \quad & x_1^{\text{new}} = x_2^{\text{old}} \\
& x_2^{\text{new}} = x_0 \\
& x_3^{\text{new}} = x_3^{\text{old}}.
\end{aligned}
$$

This gives a simple algorithm that progressively converges to the minimum by using information about the function values. Moreover, it only requires a single function evaluation per iterative step.

Figure 5.2 gives a graphical depiction of this local iteration process. The convergence is typically extremely fast once you can find a good neighbourhood to work in. However, the method is not guaranteed to converge, unfortunately. Thus, great care should be used with this method. Alternatively, very good starting points must always be used. The following code implements the successive parabolic approximation.

```
x1 = -1.5; x2 = -1; x3 = -0.5  # Initial guesses

f1 = x1**4 + 10 * x1 * np.sin(x1**2)
f2 = x2**4 + 10 * x2 * np.sin(x2**2)
f3 = x3**4 + 10 * x3 * np.sin(x3**2)

for j in range(100):
    x0 = (x1 + x2) / 2 - ((f2 - f1) * (x3 - x1) * (x3 - x2))
                    / (2 * ((x2 - x1) * (f3 - f2)
                            - (f2 - f1) * (x3 - x2)))

    if x0 > x2:
        x1 = x2; f1 = f2; x2 = x0
        f2 = x0**4 + 10 * x0 * np.sin(x0**2)
    else:
        x3 = x2; f3 = f2; x2 = x0
        f2 = x0**4 + 10 * x0 * np.sin(x0**2)

    if abs(x2 - x3) < 10**(-6) or abs(x2 - x1) < 10**(-6):
         break
```

This algorithm converges to the solution in less than half the iterations of the golden section search. However, it is easy to show that if the initial guesses are changed, then the minimization will simply not work.

5.1.3 fminbnd

Python has a built-in one-dimensional search algorithm where the function and the interval are specified. The function **fminbnd** is based upon a combination of the golden section search and successive parabolic search. Integrated together they form an effective technique for finding minima. The following code gives an example of how to execute the function:

```
x=fminbnd('x^2*cos(x)',3,4)
```

Here the left and right values of the search interval are given by $x = 3$ and $x = 4$, respectively. In this case, the function $f(x) = x^2 \cos(x)$ was found to have a minimum at $x = 3.6436$.

5.2 Unconstrained Optimization (Derivative Methods)

The methods of the previous section do not utilize any derivative information about the objective function of interest. However, in many cases, the explicit functional form to be considered for minimization is known, thus suggesting that derivatives may help in finding optimization solutions. Indeed, in simple one-dimensional problems for finding the minimum of $f(x) = 0$, it is well known that a minimum is found when $f'(x) = 0$ and $f''(x) > 0$. A maximum can be found when $f'(x) = 0$ and $f''(x) < 0$. Such ideas are easily integrated into an optimization algorithm.

To begin, we generalize the concept of a minimum or maximum, i.e. an extremum for a multi-dimensional function $f(\mathbf{x})$. At an extremum, the gradient must be zero so that

$$\nabla f(\mathbf{x}) = 0\,. \tag{5.2.1}$$

Unlike the one-dimensional case, there is no simple second-derivative test to apply to determine if the extremum point is a minimum or maximum. The idea behind gradient descent, or steepest descent, is to use the derivative information as the basis of an iterative algorithm that progressively moves closer and closer to the minimum point $f(\mathbf{x}) = 0$.

To illustrate how to proceed in practice, consider the simple example two-dimensional surface

$$f(x, y) = x^2 + 3y^2 \tag{5.2.2}$$

which has the minimum located at the origin $(x, y) = 0$. The gradient for this function can be easily computed

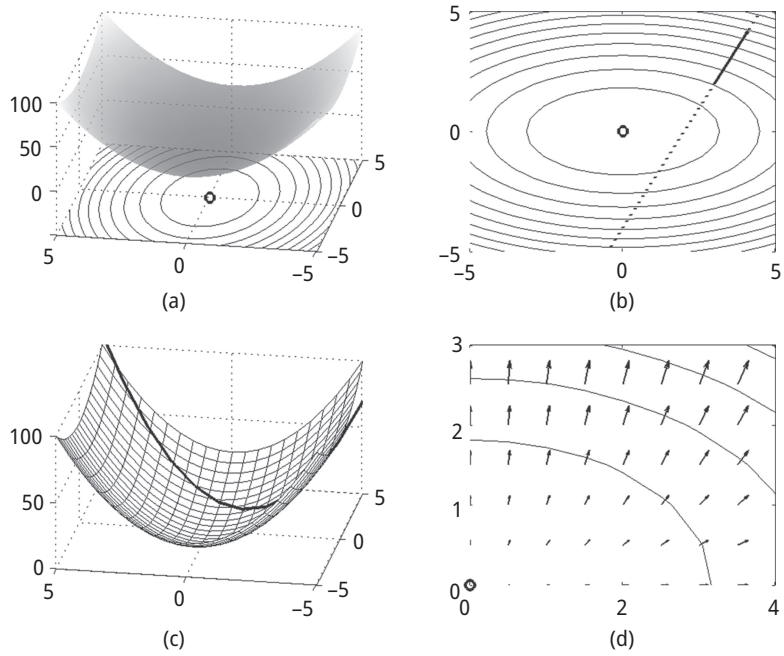


Figure 5.3 Graphical depiction of the gradient descent algorithm. The surface $f(x, y) = x^2 + 3y^2$ is plotted along with its contour lines. In (a), the surface is plotted along with the contour plot beneath. In (b), the gradient is calculated at the point $(x, y) = (3, 2)$. The gradient, which points away from the minimum, is plotted with the dark bolded line while the gradient line through $(x, y) = (3, 2)$ is plotted with a dotted line. The gradient descent moves along the steepest line of descent as can be shown in panel (c). Once the bottom of the descent curve is reached, a new descent path is picked. Panel (d) shows the overall gradient of the surface in the upper right quadrant.

$$\nabla f(\mathbf{x}) = \frac{\partial f}{\partial x}\hat{\mathbf{x}} + \frac{\partial f}{\partial y}\hat{\mathbf{y}} = 2x\hat{\mathbf{x}} + 6y\hat{\mathbf{y}} \tag{5.2.3}$$

where $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are unit vectors in the x- and y-directions, respectively.

Figure 5.3 illustrates the steepest descent (gradient descent) algorithm. At the initial guess point, the gradient $\nabla f(\mathbf{x})$ can be computed. This gives the steepest descent towards the minimum point of $f(\mathbf{x})$, i.e. the minimum is located in the direction given by $-\nabla f(\mathbf{x})$. Note that the gradient does not point at the minimum, but rather gives the steepest path for minimizing $f(\mathbf{x})$. The geometry of the steepest descent suggests the construction of an algorithm whereby the next point of iteration is picked by following the steepest descent so that

$$\boldsymbol{\xi}(\tau) = \mathbf{x} - \tau\nabla f(\mathbf{x}) \tag{5.2.4}$$

where the parameter τ dictates how far to move along the gradient descent curve. Figure 5.3(c) shows that the gradient descent curve gives a descent path that eventually

reaches bottom and starts to go back up again. In gradient descent, it is crucial to determine when this bottom is reached so that the algorithm is always *going downhill* in an optimal way. This requires the determination of the correct value of τ in the algorithm.

To compute the value of τ, consider the construction of a new function

$$F(\tau) = f(\boldsymbol{\xi}(\tau)) \tag{5.2.5}$$

which must be minimized now as a function of τ. This is accomplished by computing $dF/d\tau = 0$. Thus, one finds

$$\frac{\partial F}{\partial \tau} = -\nabla f(\boldsymbol{\xi}) \nabla f(\mathbf{x}) = 0\,. \tag{5.2.6}$$

The geometrical interpretation of this result is the following: $\nabla f(\mathbf{x})$ is the gradient direction of the current iteration point and $\nabla f(\boldsymbol{\xi})$ is the gradient direction of the future point, thus τ is chosen so that the two gradient directions are orthogonal.

For the example given above with $f(x, y) = x^2 + 3y^2$, we can easily compute this condition as follows:

$$\boldsymbol{\xi} = \mathbf{x} - \tau \nabla f(\mathbf{x}) = (1 - 2\tau)x\ \hat{\mathbf{x}} + (1 - 6\tau)y\ \hat{\mathbf{y}}\,. \tag{5.2.7}$$

This is then used to compute

$$F(\tau) = f(\boldsymbol{\xi}(\tau)) = (1 - 2\tau)^2 x^2 + 3(1 - 6\tau)^2 y^2 \tag{5.2.8}$$

whereby its derivative with respect to τ gives

$$F'(\tau) = -4(1 - 2\tau)x^2 - 36(1 - 6\tau)y^2\,. \tag{5.2.9}$$

Setting $F'(\tau) = 0$ then gives

$$\tau = \frac{x^2 + 9y^2}{2x^2 + 54y^2} \tag{5.2.10}$$

as the optimal descent step length. This gives us all the information necessary to perform the steepest descent search for the minimum of the given function. As is clearly evident, this descent search algorithm based upon derivative information is very much like Newton's method for root finding both in one dimension as well as higher dimensions. Moreover, the gradient descent algorithm is the core algorithm of advanced iterative solvers such as the bi-conjugate gradient descent method (**bicgstab**) and generalized method of residuals (**gmres**).

In what follows, we develop a Python code to perform the gradient descent search for the function $f(x, y) = x^2 + 3y^2$.

```
x = [3]; y = [2]
f = [x[0]**2 + 3 * y[0]**2]

for j in range(100):
    tau = (x[j]**2 + 9 * y[j]**2) / (2 * x[j]**2 + 54 * y[j]**2)
    x.append((1 - 2 * tau) * x[j])
    y.append((1 - 6 * tau) * y[j])
    f.append(x[j+1]**2 + 3 * y[j+1]**2)

    if abs(f[j+1] - f[j]) < 10**(-6):
        break
```

The above algorithm converges in only 11 iteration steps to the minimal solution (see Fig. 5.4). Interestingly enough, if a simple radially symmetric function is considered, then the gradient descent converges in a single iteration since the gradient descent would point directly at the minimum. As with other iterative schemes of this sort, including the root finding algorithms based upon the Newton method, convergence to the solution often depends on a user's ability to provide a good initial guess for the minimal value.

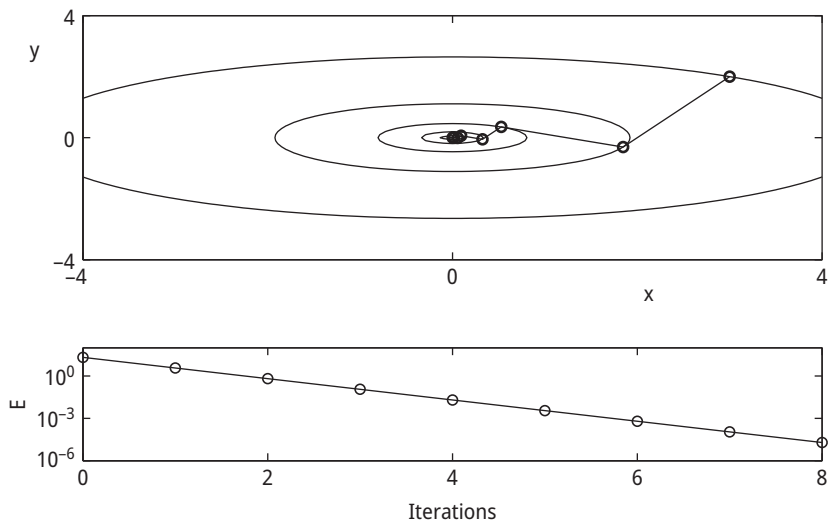


Figure 5.4 Gradient descent algorithm applied to the function $f(x, y) = x^2 + 3y^2$. In the top panel, the contours are plotted for each successive value (x, y) in the iteration algorithm given the initial guess $(x, y) = (3, 2)$. Note the orthogonality of each successive gradient in the steepest descent algorithm. The bottom panel demonstrates the rapid convergence and error (E) to the minimum (optimal) solution.

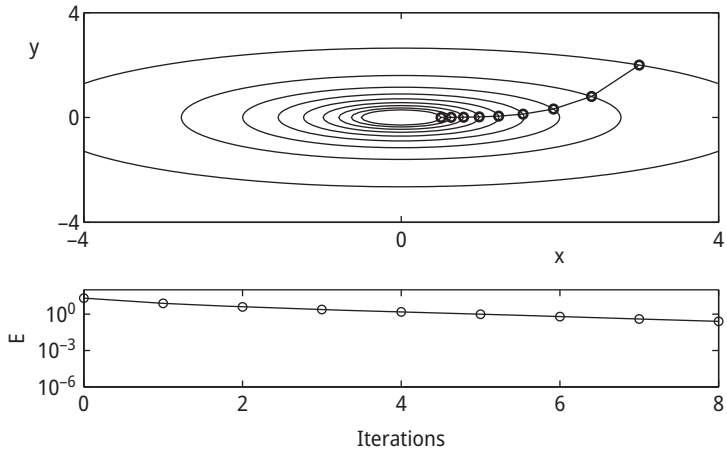


Figure 5.5 Gradient descent algorithm applied to the function $f(x, y) = x^2 + 3y^2$ with a fixed $\tau = 0.1$. In the top panel, the contours are plotted for each successive value (x, y) in the iteration algorithm given the initial guess $(x, y) = (3, 2)$. In this case, successive gradients are no longer orthogonal. The convergence and error (E) to the minimum (optimal) solution is slower with this line search method of a fixed value of τ.

The above algorithm assumes a line search algorithm to find an optimal value of τ. In particular, the value of τ picked here is optimal in the sense that a given line search is conducted so that the minimum of the gradient direction is picked as the next iteration point. However, this is not a requirement. In fact, one can simply choose a fixed value of τ for stepping forward along the gradient direction. Figure 5.5 demonstrates this case for $\tau = 0.1$. This method also converges to the solution, however at a much slower rate. Such a method may be favourable in a case where the steepest descent algorithm *zig-zags* a large amount in trying to make the projective steps orthogonal. This can happen in cases where *long-valley* type structures exist in the function we are trying to minimize.

5.2.1 minimize

Although not based upon gradient descent algorithms, the **minimize** algorithm in Python is a generic, nonlinear unconstrained optimization method based upon the Nelder–Mead simplex method [25]. We have already used this method as a means of doing nonlinear curve fitting. In that case, the objective function was the E_2 error that was to be minimized. As a second example of this technique, consider once again a set of data that we wish to fit with the function $f(x) = A\cos(Bx) + C$, where A, B, and C are the variables to be chosen for minimizing the error. Our objective function in this case is the least-square error $E_2 = \sqrt{(1/N)\sum |f(x_j) - y_j|^2}$. Thus, we only need to consider minimizing $\sum |f(x_j) - y_j|^2$ with respect to A, B, and C to achieve our goal.

The following code performs the optimization process with initial guesses given by $(A, B, C) = (12, \pi/12, 63)$.

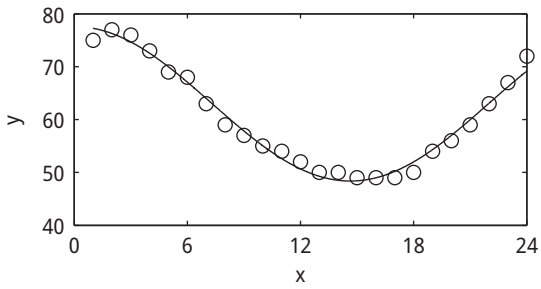


Figure 5.6 Minimization algorithm **minimize** used for curve fitting to a nonlinear function. The dots are the original data points and the solid line is the least-square fit. In this case, the least-square error E_2 is the objective function.

```
from scipy.optimize import minimize

def tempfit(c, x, y):
    e2 = np.sqrt(np.sum((c[0] * np.cos(c[1] * x) + c[2] - y) ** 2) / 24)
    return e2

x = np.arange(1, 25)
y = np.array([75, 77, 76, 73, 69, 68, 63, 59, 57, 55, 54, 52,
              50, 50, 49, 49, 49, 50, 54, 56, 59, 63, 67, 72])

c0 = [12, np.pi/12, 63]  # Initial guess
result = minimize(tempfit, c0, args=(x, y), method='Nelder-Mead')
optimal_params = result.x
```

The algorithm will rapidly converge to new values of the vector $\mathbf{c}$, which contains the updated and optimal value of A, B, and C. To plot the results and compare the fit (see Fig. 5.6). This example illustrates both the construction of an objective function as well as the implementation of one of the most important unconstrained optimization tools that is available in Python. Critical to success in this algorithm is the initial guess used for the optimal (minimal) solution.

5.3 Linear Programming

We now come to perhaps the most import aspect in terms of application: optimization with constraint. This is still a highly active area of research and many methods exist, which exploit the underlying nature of the problem being considered. Here, we will limit our discussion to a classic problem known as a *linear program*. A linear program is an optimization problem

in which the objective function is linear in the unknown and the constraints consist of linear inequalities and equalities.

To illustrate the linear programming concept, the so-called *standard form* will first be considered.

$$\begin{aligned} \text{minimize} \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \\ \text{and} \quad & x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0 \end{aligned} \tag{5.3.1}$$

which can be written in a much more elegant form via vector and matrix notation

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \text{ and } \mathbf{x} \geq 0\,. \end{aligned} \tag{5.3.2}$$

Thus, given the matrix $\mathbf{A}$ and the vectors $\mathbf{b}$ and $\mathbf{c}$, the goal is to find the vector $\mathbf{x}$ that minimizes the linear objective function given by $\mathbf{c}$.

Of course, not all linear optimization problems come directly in this form. But they can be transformed to the standard form by simple techniques.

5.3.1 Slack variables

Consider instead the following related problem which has inequality constraints instead of equality constraints.

$$\begin{aligned} \text{minimize} \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\ \text{and} \quad & x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0. \end{aligned} \tag{5.3.3}$$

This problem is no longer in the standard form. However, it can be easily put into the standard form by introducing *slack variables* so that the inequalities can be made into equalities. Thus, we transform the problem to the following:

$$
\begin{aligned}
\text{minimize} \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
\text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + y_1 = b_1 \\
& a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + y_2 = b_2 \\
& \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + y_m = b_m \\
\text{and} \quad & x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0 \\
\text{and} \quad & y_1 \geq 0, y_2 \geq 0, \ldots, y_n \geq 0.
\end{aligned}
\tag{5.3.4}
$$

The introduction of the new m variables given by $\mathbf{y}$ now sets the problem to be in standard form. In particular, the new matrix $\bar{\mathbf{A}}$ associated with the problem is now of the special form $\bar{\mathbf{A}} = [\mathbf{A}, \mathbf{I}]$, where $\mathbf{I}$ is the identify matrix, and the new vector $\bar{\mathbf{x}}$ to be solved for is $\bar{\mathbf{x}} = [\mathbf{x}, \mathbf{y}]$.

Other techniques exist to transform a linear optimization problem. If the inequalities are the opposite to the above, then *surplus* variables are introduced. If some of the unknown variables are actually not required to be positive, then they can be transformed using *free variables* [26]. Python's own built-in linear programming subroutine accepts a different form than the standard form, saving you the work of transforming it to this specific form.

Any vector $\mathbf{x}$ that satisfies the constraints of (5.3.2) is a *feasible solution*. A feasible solution is called an *optimal solution* if, in addition, the objective function in (5.3.2), i.e. $\mathbf{c}^T\mathbf{x}$, is minimal in comparison with all other feasible solutions. A *basic feasible solution* is one for which $m - n$ of the variables $\mathbf{x}$ are zero, i.e. the number of nonzero solution elements is commensurate with the number of constraints. This leads to an important theorem of linear programming [26]:

Fundamental theorem of linear programming: *Given a linear program in the standard form (5.3.2) where* $\mathbf{A}$ *is an* $m \times n$ *matrix of rank* m,

(i) *if there exists a feasible solution, there is a basic feasible solution.*

(ii) *if there is an optimal feasible solution, there is an optimal basic feasible solution.*

The goal of linear programming is to find the optimal basic feasible solution of (5.3.2). As one might imagine, there have been a great number of mathematical techniques developed to solve this critically important problem [26]. Here we will consider how to think about (5.3.2) graphically and then Python's linear programming function will be introduced.

5.3.2 A graphical interpretation

To illustrate the idea of *feasible solutions*, **basic feasible solutions**, and transforming to the *standard form*, consider the following simple example

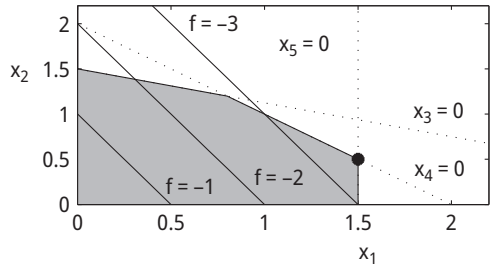


Figure 5.7 Graphical representation of the feasible region (shaded) given the constraints (5.3.5). The constraints are represented in terms of the slack variables. The objective function $f = \mathbf{c}^T\mathbf{x}$ is evaluated along contour lines. Thus, the linear program seeks to minimize f while satisfying the constraints, i.e. the linear program would identify the point $(x_1, x_2) = (1.5, 0.5)$ as the optimal basic feasible solution.

$$\begin{aligned} \text{minimize} \quad & -2x_1 - x_2 \\ \text{subject to} \quad & x_1 + (8/3)x_2 \leq 4 \\ & x_1 + x_2 \leq 2 \\ & 2x_1 \leq 3 \\ \text{and} \quad & x_1 \geq 0, x_2 \geq 0. \end{aligned} \tag{5.3.5}$$

The idea is to first write this in the standard form by introducing three slack variables to handle the three constraint inequalities. Thus, we have the new problem in standard form:

$$\begin{aligned} \text{minimize} \quad & -2x_1 - x_2 \\ \text{subject to} \quad & x_1 + (8/3)x_2 + x_3 = 4 \\ & x_1 + x_2 + x_4 = 2 \\ & 2x_1 + x_5 = 3 \\ \text{and} \quad & x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0 \end{aligned} \tag{5.3.6}$$

where the slack variables are x_3, x_4, and x_5. Given the three constraints, it is ideal to find a *basic feasible solution* that has two of the five variables set to zero.

To begin discussing the solution of this problem, we first consider the region of feasibility solutions. Thus, we consider $x_3 = 0$ in the first constraint, $x_4 = 0$ in the second constraint, and $x_5 = 0$ in the third constraint. Figure 5.7 demonstrates the feasibility region associated with this example. Once the feasibility solution is found, our objective is to minimize the objective function

$$\min f(x_1, x_2) = \min \mathbf{c}^T\mathbf{x} = -2x_1 - x_2\,. \tag{5.3.7}$$

Figure 5.7 also demonstrates the lines of constant f. Note that the value of f decreases as the line of constant f is pushed to the right. The point $(x_1, x_2) = (1.5, 0.5)$ is the furthest point in the feasible region that one can push to the right; thus, it is the optimal solution. Moreover, it is a basic optimal solution since $x_4 = x_5 = 0$ at this solution point.

5.3.3 linprog

Of course, what is desired is a systematic way to find the basic optimal solution. In the example given previously, it was simple to see from plotting alone where the optimal solution would be. However, in higher-dimensional problems, the aid of such graphical techniques is rarely available. Thus, algorithmic constructs for finding feasible solutions, and then iterating towards the optimal feasible solution, are of primary importance. Such methods have been developed; for example, the simplex method and/or interior point methods. These fall outside the scope of this book, but they can be followed up on in the literature [26].

Here, Python's linear program subroutine, **linprog**, will be considered. This is an extremely powerful tool for solving linear programming problems. The form of the linear program used by Python is slightly different from the standard form. In particular, Python will solve the following problem:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x} && (5.3.8)\\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \bar{\mathbf{A}}\mathbf{x} = \bar{\mathbf{b}} \\ & \mathbf{x}_- \leq \mathbf{x} \leq \mathbf{x}_+ && (5.3.9) \end{aligned}$$

where $\mathbf{x}_-$ and $\mathbf{x}_+$ are lower and upper bounds on the values of $\mathbf{x}$, respectively. Note that in this formulation, the equality and inequality constraints are separated. Python automatically formulates the slack/surplus variables for you.

The example given previously can be rewritten as

$$\begin{aligned} \text{minimize} \quad & -2x_1 - x_2 \\ \text{subject to} \quad & x_1 + (8/3)x_2 \leq 4 \\ & x_1 + x_2 \leq 2 \\ & 2x_1 \leq 3 && (5.3.10)\\ & -x_1 \leq 0 && (5.3.11)\\ & -x_2 \leq 0\,. \end{aligned}$$

In the matrix form as required by (5.3.8), one would then have

$$\mathbf{A} = \begin{bmatrix} 1 & 8/3 \\ 1 & 1 \\ 2 & 0 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -2 \\ -1 \end{bmatrix}. \tag{5.3.12}$$

Note that in this case, there are no equality constraints so that $\bar{\mathbf{A}}$ and $\bar{\mathbf{b}}$ do not need to be defined. Nor do we need to define bounds on the solution. The Python code for implementing this linear program is as follows:

```
from scipy.optimize import linprog

c = np.array([-2, -1])
A = np.array([[1, 8/3], [1, 1], [2, 0], [-1, 0], [0, -1]])
b = np.array([4, 2, 3, 0, 0])

result = linprog(c, A_ub=A, b_ub=b)
print("Optimal solution:", result.x)
```

This produces the optimal solution $(x_1, x_2) = (1.5, 0.5)$.

5.3.4 Open source optimization packages: cvx

Of course, linear programming can be quite restrictive since it is, in fact, limited to linear objective functions and linear constraints. There are methods available for *nonlinear programming* [26], however, they are beyond the scope of this book. Thankfully, there are a number of open source convex optimization codes that can be downloaded from the Internet. In the compressive sensing chapter to come, a convex optimization package is used that can be directly implemented with Python: http://cvxr.com/cvx/. This is one of several codes that can be downloaded that use state-of-the-art optimization techniques that go far beyond both the constraints of linear programming.

5.4 Simplex Method

Before moving on from the linear programming method, a key issue must be addressed: From a feasible solution, how can new feasible solutions be generated that are more optimal? Indeed, how can one find the optimal solution, which is the solution of the linear programming algorithm. Here, the *simplex method* is discussed, which was developed by G.B. Dantzig in 1948. As is expected, the simplex method is a systematic iterative technique which aims to take a given *basic feasible solution* to another *basic feasible solution* for which the objective function is smaller.

Consider once again the linear program in standard form:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \\ \text{and} \quad & x_1 \geq 0, x_2 \geq 0, \cdots, x_n \geq 0 \end{aligned} \tag{5.4.1}$$

where we have now represented the objective function as $f(\mathbf{x})$.

First, we can easily consider the constraint conditions and feasibility. Specifically, the constraint equations are simply $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $m \times n$ matrix, where $m < n$. Thus, the constraint system is underdetermined, and there are an infinite number of possible solutions (see Fig. 5.7 which shows the entire (shaded) region of infinite solutions). Thus, since we are guaranteed a solution to the underdetermined system, we are guaranteed a feasible solution. But once this feasible solution is found, we can easily put it into the form of a *basic feasible solution* by converting it (via Gaussian elimination type techniques) to the canonical form:

$$\begin{aligned} & x_1 + y_{1,m+1}x_{m+1} + y_{1,m+2}x_{m+2} + \cdots + y_{1,n}x_n = y_{10} \\ & x_2 + y_{2,m+1}x_{m+1} + y_{2,m+2}x_{m+2} + \cdots + y_{2,n}x_n = y_{20} \\ & \vdots \\ & x_m + y_{m,m+1}x_{m+1} + y_{m,m+2}x_{m+2} + \cdots + y_{m,n}x_n = y_{m0}\,. \end{aligned} \tag{5.4.2}$$

Once in the canonical form, a basic feasible solution is found, where

$$x_1 = y_{10}, x_2 = y_{20}, x_m = y_{m0}, \quad \text{and} \quad x_{m+1} = 0, x_{m+2} = 0, \ldots, x_n = 0\,. \tag{5.4.3}$$

This canonical solution is also a basic feasible solution. The variables $x_1, x_2, \ldots, x_m$ are called *basic* and the variables $x_{m+1}, x_{m+2}, \ldots, x_n$ are called *nonbasic*.

Here is the fundamental question to ask: Do we have the right basic and nonbasic variables? Specifically, what if there is a variable x_p, where p is from somewhere in $m+1$ to n, such that it would be a better choice as a basic variable, i.e. it would give a more optimal solution where the objective function is smaller. The simplex method fundamentally is concerned with making basic those variables that, in fact, give an optimal solution. Thus, an iteration procedure must be created to perform such an action.

To move forward, the simplex *tableau* is created for the above basic feasible solution. Thus, we can write this in a more shorthand notation as:

$$\begin{array}{cccccccccc} 1 & 0 & \cdots & 0 & y_{1,m+1} & y_{1,m+2} & \cdots & y_{1,n} & y_{1,0} \\ 0 & 1 & \cdots & 0 & y_{2,m+1} & y_{2,m+2} & \cdots & y_{2,n} & y_{2,0} \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & \cdots & 1 & y_{m,m+1} & y_{m,m+2} & \cdots & y_{m,n} & y_{m,0}. \end{array} \tag{5.4.4}$$

The purpose in writing it in this form is that the operations which will be performed for the simplex method are much like those in Gaussian elimination. In particular, it is often advantageous to switch basic and nonbasic variables, thus necessitating column and row reductions to achieve this goal.

Here is the critical observation, and the fundamental point, of the simplex method. Although it is natural to use the basic solutions from the computed tableau above, it is also clear that arbitrary values of $x_{m+1}, x_{m+2}, \ldots, x_n$ can be chosen. Recall that this is an underdetermined system, so an infinite number of solutions are allowed, including those with nontrivial nonbasic variables. If these nontrivial basic variables are chosen arbitrarily, then the above tableau gives the following values of the basic variables:

$$\begin{array}{lcl} x_1 & = & y_{10} - \sum_{j=m+1}^{n} y_{1,j} x_j \\ x_2 & = & y_{20} - \sum_{j=m+1}^{n} y_{2,j} x_j \\ . & . & . \\ . & . & . \\ x_m & = & y_{m0} - \sum_{j=m+1}^{n} y_{m,j} x_j \, . \end{array} \tag{5.4.5}$$

Of course, this is a trivial observation. But it has a profound impact when considering the objective function

$$\begin{aligned} f &= c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\ &= f_0 + (c_{m+1} - f_{m+1}) x_{m+1} + (c_{m+2} - f_{m+2}) x_{m+2} + \cdots + (c_n - f_n) x_n \end{aligned} \tag{5.4.6}$$

where

$$f_j = y_{1,j} c_1 + y_{2,j} c_2 + \cdots + y_{m,j} c_m \tag{5.4.7}$$

with $m + 1 \leq j \leq n$. The critical observation is that this formulation gives the value of the objective function $f(\mathbf{x})$ in terms of the nonbasic variables $x_{m+1}, x_{m+2}, \ldots, x_n$. Thus, from this we can determine if there is an advantage to switching basic to nonbasic variables in order to minimize the objective function. Specifically, if any $(c_j - f_j) < 0$ in the above formula, then the objective function will be lowered. The following theorem then applies:

Theorem (Improvement of basic feasible solution): *Given a nondegenerate basic feasible solution with corresponding objective function f_0, suppose that for some j there holds $(c_j - z_j) < 0$. Then there is a feasible solution with objective function value $f < f_0$. If the column $\mathbf{a}_j$ can be substituted for some vector in the original basis to yield a new basic feasible solution, this new solution will have $f < f_0$. If $\mathbf{a}_j$ cannot be substituted to yield a basic*

feasible solution, then the solution set is unbounded and the objective function can be made arbitrarily small (toward minus infinity).

The above theorem is the basis for the simplex method. Thus, from a given basic feasible solution, it only remains to identify $(c_j - z_j) < 0$ and use pivoting and row reduction techniques to swap basic and nonbasic solutions so that a new, and smaller, objective function is achieved. This process is continued until no $(c_j - z_j) < 0$ remain. In fact, the following optimality theorem then holds:

Theorem (Optimality Condition Theorem): *If for some basic feasible solution* $(c_j - z_j) \geq 0$ *for all* j*, then the solution is optimal.*

Armed with the above two theorems, the simplex method can be constructed and a termination point reached in the iteration method. Note that just like Gaussian elimination, which involves the same basic procedures in row and column reductions and manipulations, the larger the matrix, the greater the time in computation.

To illustrate how the actual process is achieved, consider the objective function $f(\mathbf{x}) = \sum x_j$ and the following simplex tableau:

$$\begin{array}{rrrrrrr} 1 & 0 & 0 & 2 & 4 & 6 & 4 \\ 0 & 1 & 0 & 1 & 2 & 3 & 3 \\ 0 & 0 & 1 & -1 & 2 & 1 & 1 \end{array} \tag{5.4.8}$$

where the first six columns correspond to the coefficients of x_1, x_2, x_3, x_4, x_5, and x_6. The final column is the coefficients of the constraint vector $\mathbf{b}$. The basic feasible solution in this case is

$$\mathbf{x} = (4, 3, 1, 0, 0, 0) \quad \text{with} \quad f = \sum x_j = 8. \tag{5.4.9}$$

Now suppose we elect to bring the fourth column $\mathbf{a}_4$ into the basis, i.e. make it a basic versus nonbasic variable. Then it is necessary to determine which element in the fourth column is the appropriate pivot. The following three ratios are computed

$$b(1)/y_{1,4} = 4/2 = 2, \;\; b(2)/y_{2,4} = 3/1 = 3, \;\; b(3)/y_{3,4} = 1/-1 = -1. \tag{5.4.10}$$

The idea is to choose the smallest positive pivot, thus the pivot point will be the $b(1)/y_{1,4} = 2$ term, and the pivoting happens about the first row, fourth column. As with Gaussian elimination, the goal is to make the other elements of the fourth column zero, which can be achieved by adding and subtracting appropriately scaled rows. This yields the new simplex tableau:

$$\begin{array}{rrrrrrr} 1/2 & 0 & 0 & 1 & 2 & 3 & 2 \\ -1/2 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1/2 & 0 & 1 & 0 & 4 & 4 & 3 \end{array} \tag{5.4.11}$$

which has the basic feasible solution

$$\mathbf{x} = (0, 1, 3, 2, 0, 0) \quad \text{with} \quad f = \sum x_j = 6\,. \tag{5.4.12}$$

This simple example shows that the objective function was reduced from 8 to 6 simply by switching a basic with nonbasic variable.

More generally, the simplex algorithm proceeds as follows:

(1) Form a simplex tableau from the initial basic feasible solution and compute the $(c_j - f_j)$.

(2) If each $(c_j - f_j) \geq 0$, stop the algorithm since the basic feasible solution is optimal.

(3) Select the jth column for which $(c_j - f_j) < 0$ is the least negative. This column will be made into a basic variable.

(4) Determine all the potential pivot values by evaluating y_{k0}/y_{kj} for $y_{kj} > 0$ and $k = 1, 2, \ldots, m$. If no $y_{kj} > 0$, then stop as the problem is unbounded. Otherwise, select p as the index k corresponding to the minimum ratio.

(5) Pivot on the pjth element, updating all rows including the last. Return to the first step (1).

This gives the basic outline of the technique. Of course, just like Gaussian elimination, certain problems can arise in the pivoting process, including if there is degeneracy in the system. There are numerous techniques and algorithm improvements for the simplex method, and one is encouraged to follow these up in the literature [26].

5.5 Genetic Algorithms

Other methods developed for optimization problems are the so-called *genetic algorithms*, which are a subset of evolutionary algorithms. The principle is quite simple and mirrors what is perceived to occur in evolution and/or genetic mutations. In particular, given a set of feasible trial solutions (either constrained or unconstrained), the objective function is evaluated. In the language of genetic algorithms, the objective function is now called the *fitness function*. The idea is to keep those solutions that give the minimal value of the objective function and mutate them in order to try and do even better. Thus beneficial mutations, in the sense of giving a better minimization, are kept while those that perform poorly are thrown away, i.e. survival of the fittest. This process is repeated through a prescribed number of iterations, or *generations*, with the idea that better and better fitness function values are generated via the mutation process.

To be more precise about the genetic algorithm structure, consider the unconstrained optimization problem with the objective function

$$\min f(\mathbf{x}) \tag{5.5.1}$$

where $\mathbf{x}$ is an n-dimensional vector. Suppose that m initial guesses are given for the values of $\mathbf{x}$ so that

$$\text{guess } j \text{ is } \mathbf{x}_j \,. \tag{5.5.2}$$

Thus, m solutions are evaluated and compared with each other in order to see which of the solutions generate the smallest objective function since our goal is to minimize it. We can order the guesses so that the first $p < m$ give the smallest values of $f(\mathbf{x})$. Arranging our data, we then have

$$\begin{aligned} \text{keep} \quad & \mathbf{x}_j \; j = 1, 2, \ldots, p \\ \text{discard} \quad & \mathbf{x}_j \; j = p+1, p+2, \ldots, m. \end{aligned} \tag{5.5.3}$$

Since the first p solutions are the best, these are kept in the next generation. In addition, we now generate $m - p$ new trial solutions that are randomly mutated from the p best solutions. This process is repeated through a finite number of iterations with the hope that convergence to the optimal solution is achieved.

Interestingly, there are really no theorems about the convergence of such a technique, so one may wonder why it should be considered at all when guaranteed convergence can be achieved with alternative algorithms. The use of such algorithms is due to a few key advantages that are difficult to find elsewhere. First, many iteration schemes can *get stuck* in the iteration process for a variety of reasons. Thus, convergence is extremely slow, or only a local minimum can be found. In the mutation process of the genetic algorithm, there exists the possibility of moving well beyond these pernicious points so that the iteration can continue moving towards the optimal solution. Second, in all that has been considered thus far, an optimization problem can be neatly packaged as a set of constraints with an objective function. However, suppose the problem is sufficiently complex so that nonlinear constraints exist and the methods developed previously simply no longer hold. Alternatively, what if the objective function to be minimized is only computable after a larger simulation has been performed? Thus, the idea is to choose the parameters of this larger simulation based upon the genetic algorithm itself and its ability to minimize the objective function.

To demonstrate the concept, consider the example in Fig. 5.6 which was solved using the **minimize** algorithm. In this case, the function form $f(x) = A\cos(Bx) + C$ is assumed and the fitness function (objective function) is the E_2 error. Here, a genetic algorithm will be developed that will search for the optimal solution using a set of initial guesses followed by mutations of the best solutions. We begin by defining some initial parameters for the genetic algorithm. In particular, 200 generations will be run with 50 trial solutions. Only the top 10 best solutions will be kept and mutated at the next generation. As before with **minimize**, an initial guess of $A = 12$, $B = pi/12$, and $C = 60$ will be used.

```
x = np.arange(1, 25)
y = np.array([75, 77, 76, 73, 69, 68, 63, 59, 57, 55, 54, 52,
              50, 50, 49, 49, 49, 50, 54, 56, 59, 63, 67, 72])

m = 1000  # number of generations
n = 80   # number of trials
n2 = 40  # number of trials to be kept

A = 12 + np.random.randn(n, 1)  # Initialize  A, B, C
B = np.pi / 12 + np.random.randn(n, 1)
C = 60 + np.random.randn(n, 1)

for jgen in range(m):     # Evaluate objective function
    E = np.zeros(n)
    for j in range(n):
        E[j] = np.sum((A[j] * np.cos(B[j] * x) + C[j] - y) ** 2)

    Ej_sorted = np.argsort(E)  # Sort
    Ej = Ej_sorted[:n2]

    Ak1 = A[Ej] # Get best n2 solutions
    Bk1 = B[Ej]
    Ck1 = C[Ej]

    # Generate n2 new mutations
    Ak2 = Ak1 + np.random.randn(n2, 1)
    Bk2 = Bk1 + np.random.randn(n2, 1)
    Ck2 = Ck1 + np.random.randn(n2, 1)

    # Group new n gueses
    A = np.concatenate((A[:n2], Ak2))
    B = np.concatenate((B[:n2], Bk2))
    C = np.concatenate((C[:n2], Ck2))

xx = np.arange(1, 24.01, 0.01)
yfit = (A[0] * np.cos(B[0] * xx) + C[0]).flatten()
```

Note that the algorithm takes progressively smaller mutations as the generations progress, thus the divide by **jgen**. Although a contrived example, this genetic algorithm converges nicely to the least-square solution. It should be noted, however, that this is an extremely

slow method for doing curve fitting. So this example should be thought of as illustrative only. The convergence of the scheme and the data fit can be found in Fig. 5.8. Figure 5.9 shows the error of the 50 trials at various generations of the algorithm.

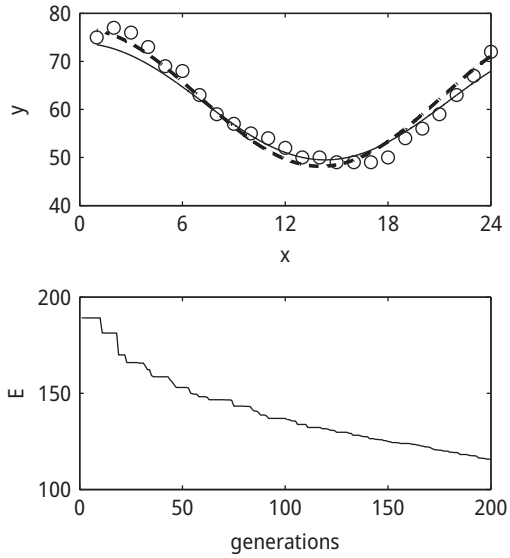


Figure 5.8 The top panel shows the data and curve fit from the genetic algorithm as developed here (solid line) and Python's genetic algorithm (dashed line). The bottom panel shows the error of the best solution at each successive generation. The error slowly converges to the same solution as **minimize**.

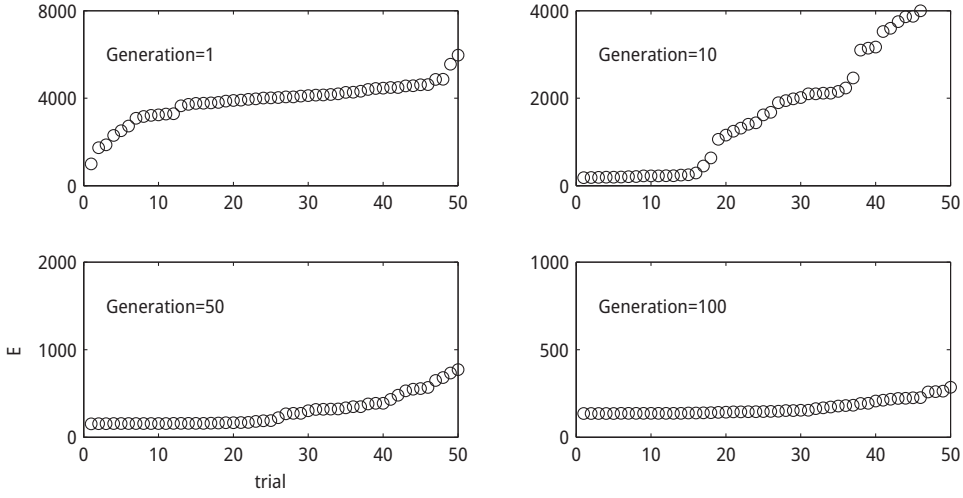


Figure 5.9 Error of the 50 trial solutions at generation 1, 10, 50, and 100. Note the convergence to the optimal solution as generations progress forward.

5.5.1 ga

Python also has a built-in genetic algorithm code that is easy to use and implement. Moreover, it can be set up to do constrained optimization problems, even with nonlinear objective functions. To begin, the **ga** algorithm is illustrated with the simple example of the curve fit just illustrated. The code for solving this problem is given by

```
from scipy.optimize import differential_evolution

def fit_line(x):
    xx = np.arange(1, 25)
    yy = np.array([75, 77, 76, 73, 69, 68, 63, 59, 57, 55, 54,
              52, 50, 50, 49, 49, 49, 50, 54, 56, 59, 63, 67, 72])
    E = np.sum((x[0] * np.cos(x[1] * xx) + x[2] - yy) ** 2)
    return E

lower = [10, np.pi/20, 50]
upper = [15, np.pi/4, 70]
result = differential_evolution(fit_line,
bounds=list(zip(lower, upper)))

print("Optimal parameters:", result.x)
```

Note that in the code, a lower and upper bound on the solution has been provided. This is equivalent to providing a good initial guess. If upper and lower bounds are not provided, then the algorithm fails completely.

More generally, the genetic algorithm as developed by Python allows for both equality and inequality constraints. Additionally, nonlinear constraints can be imposed on the problem, thus making the **ga** algorithm extremely powerful. To be specific, the following problem can be solved

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) && (5.5.4)\\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \bar{\mathbf{A}}\mathbf{x} = \bar{\mathbf{b}} \\ & g(\mathbf{x}) \le 0 \\ & \bar{g}(\mathbf{x}) = 0 \\ & \mathbf{x}_- \le \mathbf{x} \le \mathbf{x}_+ . && (5.5.5) \end{aligned}$$

The generic optimization thus allows for a nonlinear objective function $f(\mathbf{x})$ along with a set of linear equality and inequality constraints, $\mathbf{A}\mathbf{x} \le \mathbf{b}$ and $\bar{\mathbf{A}}\mathbf{x} = \bar{\mathbf{b}}$, respectively, a set

of nonlinear equality and inequality constraints, $g(\mathbf{x}) \leq 0$ and $\bar{g}(\mathbf{x}) = 0$, respectively, and upper and lower bounds, $\mathbf{x}_+$ and $\mathbf{x}_-$, respectively. A function call to the **ga** algorithm is given by

```
x = ga('fit',n,A,b,Abar,bbar,xl,xu,nonlin,options)
```

where the above constraints are placed one by one into the genetic algorithm. The options may become important as the maximum number of generations and tolerance, for instance, are set within this variable space.

6 Advanced Curve Fitting and Machine Learning

6.1 Machine Learning is Curve Fitting 134
6.1.1 Learning paradigms: unsupervised, supervised, and reinforcement 135
6.1.2 Training, testing, and cross-validation 138
6.2 Neural Networks as Curve Fits 139
6.2.1 Backpropagation (chain rule) and gradient descent 140
6.3 Learned Models: Generalization, Interpolation, and Extrapolation 144

Machine learning is now prevalent throughout the sciences and engineering. Having already covered both curve fitting and optimization techniques, one can immediately begin the consideration of machine learning methods as machine learning is ultimately a curve fitting exercise powered by the most sophisticated optimization algorithms developed to date. Unlike our simple curve fitting examples previously considered where only a few parameters were necessary to be estimated, machine learning aims to determine upwards of billions or trillions of parameters in large-scale models such as those found in speech, vision, and natural language processing. But ultimately, despite their sophistication, machine learning models are simply fancy curve fits.

6.1 Machine Learning is Curve Fitting

With some of the basic concepts of curve fitting and optimization developed in the last two chapters, we can now move on to considering the basic concepts of machine learning. Specifically, machine learning is curve fitting but with much more exotic curves (functions) than previously considered. A broader viewpoint taken here is that curve fitting maps a set of input variables ($\mathbf{X}$) to output variables ($\mathbf{Y}$) with a function or curve $f_{\boldsymbol{\theta}}(\cdot)$ with parameters $\boldsymbol{\theta}$. Thus, curve fitting is a *regression* to a mapping

$$\mathbf{Y} = f_{\boldsymbol{\theta}}(\mathbf{X}) \tag{6.1.1}$$

where the parameters $\boldsymbol{\theta}$ are found by optimization to a *goodness-of-fit* of this function to data. In a line fit, $f_{\boldsymbol{\theta}}(x) = Ax + B$, the input is the x-value, the output is the y-value, and the vector of parameters is given by $\boldsymbol{\theta} = [A, B]$. In least-square fitting, A and B are determined

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0007

by minimizing the error between the data and the curve. As was shown previously in the introduction to curve fitting, this can be done explicitly in the case of a polynomial model fit.

Machine learning builds upon the basic curve fitting framework (6.1.1) by allowing for much broader functional forms and optimization criteria. Indeed, instead of minimizing for a single objective, a number of objectives can be simultaneously specified so that

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}, \boldsymbol{\theta}) = \mathcal{L}_0 + \lambda_1 \mathcal{L}_1 + \cdots \lambda_p \mathcal{L}_p \tag{6.1.2}$$

where $\mathcal{L}_k$ is one of the p desired minimization criteria and λ_k is the weighting of this specific criterion. Determining the values of λ_k is typically done through hyper-parameter tuning of the total minimization function $\mathcal{L}$, which can be quite computationally expensive as solutions in many practical cases vary significantly as the λ_k are adjusted. Engineering the loss function $\mathcal{L}$ is one of the most critical aspects of making a machine learning model work in practice.

One of the simplest loss functions is the root-mean square error. Thus, if the model

$$\tilde{\mathbf{Y}} = f_{\boldsymbol{\theta}}(\mathbf{X}) \tag{6.1.3}$$

produces an approximation $\tilde{\mathbf{Y}}$ to the true output $\mathbf{Y}$, then the parameters $\boldsymbol{\theta}$ are determined by minimizing the loss function

$$\mathcal{L} = \|\mathbf{Y} - \tilde{\mathbf{Y}}\|_2^2. \tag{6.1.4}$$

This shows that the model $f_{\boldsymbol{\theta}}(\cdot)$ takes the input data $\mathbf{X}$ and current values of $\boldsymbol{\theta}$ in order to produce the approximate output $\tilde{\mathbf{Y}}$. An optimization procedure, such as gradient descent, is then used to update the values of $\boldsymbol{\theta}$ in order to move $\tilde{\mathbf{Y}}$ closer to the target value of $\mathbf{Y}$. Although gradient descent and root-mean square error are mentioned, the diversity of loss functions and optimization procedures are what make modern deep learning one of the most exciting and challenging fields of research.

Machine learning allows for exceptionally flexible architectures for curve fitting. Not only can standard curve fitting be done within its computational framework, but much broader abstractions can be constructed. Importantly, the three major paradigms of machine learning can be formulated within (6.1.1). This includes unsupervised, supervised (and semi-supervised), and reinforcement learning. The concept of *learning* is explicitly identified in the curve fitting process as the models selected *learn* how to represent or characterize the system of interest and its data.

6.1.1 Learning paradigms: unsupervised, supervised, and reinforcement

Learning algorithms are engineered to exploit features and patterns in data in order to often accomplish a combination of classification, prediction, and/or control. The goal of learning is driven by the specifics of the application or problem under consideration. *Supervised methods* and *unsupervised methods* are the two most dominant paradigms for learning from data, with *reinforcement learning* targeted towards control problems. In part three of this text, feature extraction algorithms will be considered in detail with the goal of aiding in machine

learning broadly. But once a set of features have been successfully extracted, the learning algorithms are employed. Supervised learning algorithms are given labelled data sets, where a set of training data is labelled by a teacher/expert/supervisor. Thus, a number of outcomes and examples of the input and output relationship of a given model is explicitly demonstrated for the algorithm. The algorithm can then use this training information to optimize the parameters $\boldsymbol{\theta}$ in the model $f_{\boldsymbol{\theta}}(\cdot)$. The trained model can then be deployed on new data. In unsupervised learning, no labels are given to the data. Rather, the goal itself is to discover patterns in the data so that feature engineering or feature extraction can be used to build an appropriate model. Thus, the algorithm itself aims to be the teacher/expert/supervisor by quantifying the clusters and patterns that are manifest in the data. Practitioners can often leverage the patterns learnt from unsupervised algorithms to build better classifiers and predictions. Semi-supervised is a hybrid method for labelling very large data sets by essentially using supervised learning to label new data. Supervised methods always perform better than unsupervised methods since explicit knowledge of the labels already exists. Reinforcement learning, which will be covered in greater depth later, is a form of supervised learning where a target objective or goal is maximized through actions taken within a given environment.

The distinction between supervised and unsupervised learning can be stated in a precise mathematical way. This will be first done for the task of *classification*, which is largely what unsupervised learning is constructed to accomplish. Supervised learning is a much more flexible paradigm that can be used for broader modelling frameworks. First, the space is defined on which data is embedded. Thus, consider

$$\mathcal{D} \subset \mathbb{R}^n \tag{6.1.5}$$

so that $\mathcal{D}$ is an open bounded set of dimension n. Further, let

$$\mathcal{D}' \subset \mathcal{D}\,. \tag{6.1.6}$$

The goal of classification is to build a classifier labelling all data in $\mathcal{D}$ given data from $\mathcal{D}'$.

We can then consider data collected from this space. Specifically, consider a set of data points $\mathbf{x}_j \in \mathbb{R}^n$, which can be used to construct the input matrix $\mathbf{X}$. The output as prescribed in (6.1.1) is given by the labels $\mathbf{y}_j$ for each point, where $j = 1, 2, \ldots, m$. The label vectors can be used to construct the output matrix $\mathbf{Y}$. Thus, the input to the model is the data $\mathbf{X}$, and the output to the model is a label encoded in $\mathbf{Y}$. Labels for the data are diverse and include numeric values, including integers, and test strings. For the purposes of illustration, the data labels considered here will be binary and either be a plus or minus one so that $\mathbf{y}_j \in \{\pm 1\}$.

For unsupervised learning, there are no labels. Thus, the input is designed to elicit an output, which is a label. Mathematically, the unsupervised task is represented as follows:

Input

$$\text{data} \quad \{\mathbf{x}_j \in \mathbb{R}^n,\ j \in Z := \{1, 2, \ldots, m\}\} \tag{6.1.7a}$$

Output

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z\}\,. \tag{6.1.7b}$$

Thus, the unsupervised learning goal is aimed at producing labels $\mathbf{y}_j$ for all the data. Generally, the data $\mathbf{x}_j$ used for training the classifier is from $\mathcal{D}'$. The classifier is then more broadly applied to the open bounded domain $\mathcal{D}$.

Using the model beyond the training data set is called generalization. The success of generalization depends largely on whether the model is interpolating or extrapolating. Extrapolation is generally a much more difficult task than interpolation. So if the training data comes from a small portion of the domain, it is often forced to extrapolate to regions of data it has not sampled from. This typically guarantees poor performance. If instead, the domain is well sampled, then the generalization task is made much easier since often one is interpolating to new data, which can work quite well. One should never forget that extrapolation and interpolation are significantly different, the former being exceptionally difficult. Thus, generalization is overall a meaningless statement unless one understands if extrapolation or interpolation is being done. In general, if generalization does not (does) work, you are probably extrapolating (interpolating).

Supervised learning has a significant advantage over unsupervised learning since labelled data is provided. The labelled data is used at the training stage in order to optimize the parameters of the model $f_{\boldsymbol{\theta}}(\cdot)$. The supervised learning classification task as a function of input and output can be stated as follows:

Input

$$\text{data} \quad \{\mathbf{x}_j \in \mathbb{R}^n,\ j \in Z := \{1, 2, \ldots, m\}\} \tag{6.1.8a}$$

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z' \subset Z\} \tag{6.1.8b}$$

Output

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z\}\,. \tag{6.1.8c}$$

In this case, a subset of the data is labelled and the missing labels are provided for the remaining data. Technically speaking, this is a semi-supervised learning task since some of the training labels are missing. For supervised learning, all the labels are known in order to build the classifier model $f_{\boldsymbol{\theta}}(\cdot)$ on $\mathcal{D}'$. The classifier is then applied to all data in $\mathcal{D}$.

The same issues of generalization need to be considered with supervised learning. Specifically, if only a small subset of the domain is sampled, then the model will be forced to extrapolate (or do extrapolatory generalization), which will with high probability lead to poor performance. If however, the space is well sampled, then the model is largely interpolating (or doing interpolatory generalization) so that the performance can be quite good and robust. The success of large language models, speech, and vision are due to the fact that exceptionally large data sets are available so that the machine learning models are operating in interpolatory fashion. Thus, exceptionally large amounts of labelled training data, sampling the full statistical distribution of the data, is often critical for success in supervised learning systems.

6.1.2 Training, testing, and cross-validation

In practice, the mathematical framework of learning requires optimization in order to find the best parameter values $\boldsymbol{\theta}$ in the model $f_{\boldsymbol{\theta}}(\cdot)$. Unlike simple curve fitting with lines and polynomials, which has a minimal number of parameters, machine learning models typically have a large number of parameters. In many cases in modern applications of deep learning, the number of parameters is greater than the number of data points m so that $\boldsymbol{\theta} \in \mathbb{R}^M$ and $M \gg m$. This gives rise to over-fitting of the data as the model has many parameters per data point. Specifically, because of such a large number of parameters, the error can often be reduced through training to extremely low-levels that are not representative of the model efficacy. Rather, the low-error is simply a representation of the over-fitting process.

To avoid over-fitting, the data must be separated into a training set and a test set in order to evaluate the quality of the model. This process is called a train-test split of the data. The goal is to randomly select a subset of the data, known as the test set, that is not used in training. The test set is only used at the conclusion of training in order to more accurately evaluate the performance of the model $f_{\boldsymbol{\theta}}(\cdot)$. In addition to the train-test split, the training data is split into a train and cross-validation subset in order to avoid over-fitting in the training itself. Thus, from a total of m points, we can make a training and test set.

$$\mathbf{Data}: \ \mathbf{x}_j, \ \mathbf{y}_j$$

training data and labels: (6.1.9a)

$$j = 1, 2, \ldots, q \quad \text{(train)} \tag{6.1.9b}$$

$$j = q+1, q+2, \ldots, p \quad \text{(cross-validate)} \tag{6.1.9c}$$

test data and labels: (6.1.9d)

$$j = p+1, p+2, \ldots, m \tag{6.1.9e}$$

Often the train-test split is 80/20, with 80% of the data used for training and 20% used for testing. One can then do another 80/20 split within the training set for cross-validation. The actual numbers depend greatly on how much data is available. As a general rule, the goal is to have a large enough training set so that the model $f_{\boldsymbol{\theta}}(\cdot)$ has thoroughly sampled the underlying statistical distribution of the data. The broader implication in this statement is that the model $f_{\boldsymbol{\theta}}(\cdot)$ is interpolatory in nature and will generally fail when data comes from outside the training distribution.

The performance of the machine learning model $f_{\boldsymbol{\theta}}(\cdot)$ is always evaluated through cross-validation. The train-test split provides the framework for such cross-validation. Importantly, as the model is trained through an optimization procedure, which iteratively updates the values of the model parameters $\boldsymbol{\theta}$, the training error will continue to be improved as over-fitting occurs. In contrast, the test error will decrease until over-fitting occurs, at which point the error on the test set increases. Typically, one stops the model training just before the test error is minimized. This is the idea behind *early stopping*, which tends to improve performance of the trained model.

6.2 Neural Networks as Curve Fits

To illustrate the broader aspects of curve fitting and its relation to machine learning, we will develop the concept of curve fitting further with neural networks. Neural networks can be thought of as exotic curves with exceptional representation capabilities. So instead of a polynomial or interpretable function like cosine or sine, neural networks offer a sophisticated structure with nodes, weights, and activation functions that can approximate sophisticated data through learning (optimization). Often this is stated in terms of their *universal approximation* capabilities [27].

Importantly, neural networks are simply maps from inputs to outputs as given by (6.1.1). The previous chapter already has shown that the inputs and outputs can be quite general, from numeric data to text strings. Neural networks are in essence compositional, nonlinear function maps. Figure 6.1 illustrates a simple neural network architecture that maps input to output through a number of *hidden layers* with hidden state variables $\mathbf{z}_k$. In the illustration shown, the neural network has p hidden layers with a specified dimension. Deep learning generally constructs neural networks that have many hidden layers that can be of high-dimension. The term *deep learning* typically refers to the number of hidden layers. In the early days of neural networks, only one or two hidden layers were considered [28]. This was due largely to the lack of computational power and the optimization algorithms required to solve larger and deeper networks in practice.

From a broader perspective, we can again consider (6.1.1) with a general compositional form. Figure 6.1 shows how to map from the hidden layer $\mathbf{z}_{k-1}$ to $\mathbf{z}_k$. Specifically, we have the mapping

$$\mathbf{z}_k = f_{\boldsymbol{\theta}_k}^{(k)}(\mathbf{z}_{k-1}) \tag{6.2.1}$$

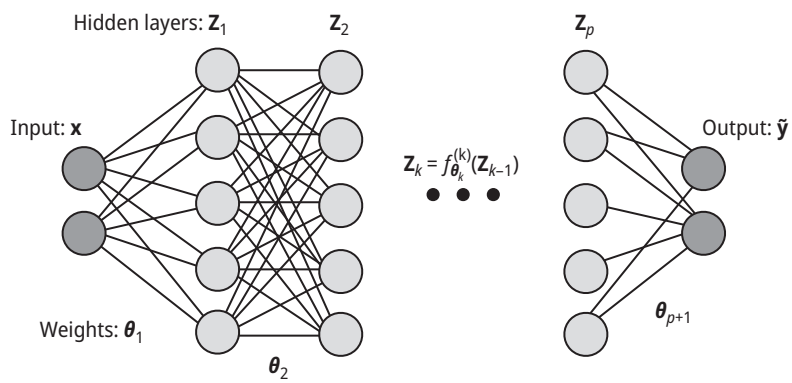


Figure 6.1 General structure of a neural network as curve fitting function. The input $\mathbf{x}$ is evaluated through a compositional framework, where the activation function $f_{\boldsymbol{\theta}_k}^{(k)}(\cdot)$ specifies the mapping function with weights $\boldsymbol{\theta}_k$. The number of layers, their dimensions, and their activation functions all play a critical role in constructing an accurate curve fitting model. All the weights $\boldsymbol{\theta} \in [\boldsymbol{\theta}_{p+1}, \boldsymbol{\theta}_p, \ldots, \boldsymbol{\theta}_2, \boldsymbol{\theta}_1]$ must be determined by an optimization procedure such as gradient descent.

where $f^{(k)}_{\boldsymbol{\theta}_k}(\cdot)$ specifies the mapping function with weights $\boldsymbol{\theta}_k$. For the first and last layer, we have

$$\mathbf{z}_1 = f^{(1)}_{\boldsymbol{\theta}_1}(\mathbf{x}) \tag{6.2.2a}$$

$$\mathbf{y} = f^{(p+1)}_{\boldsymbol{\theta}_{p+1}}(\mathbf{z}_p) \tag{6.2.2b}$$

Each layer can have a different dimension along with different functions $f^{(k)}_{\boldsymbol{\theta}_k}(\cdot)$, which are referred to as activation functions. Often the mapping (6.2.1) is represented as

$$\mathbf{z}_k = f^{(k)}(\mathbf{w}_k\mathbf{z}_{k-1} + \mathbf{b}_k) \tag{6.2.3}$$

where the weights are separated into a weight matrix acting on the current layer $\mathbf{z}_{k-1}$ with a bias $\mathbf{b}_k$ used as a bias offset, or DC component to the mapping. For convenience, the weights and biases, $\mathbf{z}_{k-1}$ and $\mathbf{b}_k$, will be just combined into $\boldsymbol{\theta}_k$.

The neural network in Fig. 6.1 can thus be represented as the following compositional function over p hidden layers:

$$\tilde{\mathbf{Y}} = f^{(p+1)}_{\boldsymbol{\theta}_{p+1}}(f^{(p)}_{\boldsymbol{\theta}_p}(\cdots f^{(2)}_{\boldsymbol{\theta}_2}(f^{(1)}_{\boldsymbol{\theta}_1}(\mathbf{X})))) \tag{6.2.4}$$

where $f^{(k)}_{\boldsymbol{\theta}_k}(\cdot)$ denotes the function with weights $\boldsymbol{\theta}_k$ that maps data from the latent space $\mathbf{Z}_{k-1}$ to $\mathbf{Z}_k$. Here, there are p total layers that are in a compositional structure to map from $\mathbf{X}$ to the output approximation $\tilde{\mathbf{Y}}$. The set of all weights are represented by

$$\boldsymbol{\theta} \in [\boldsymbol{\theta}_{p+1}, \boldsymbol{\theta}_p, \ldots, \boldsymbol{\theta}_2, \boldsymbol{\theta}_1]. \tag{6.2.5}$$

The compositional function (6.2.4) is our curve to be fit to the data. And the parameters of the fit, given by (6.2.5), are what must be determined. This is a high-dimensional optimization problem since $\boldsymbol{\theta}$ is typically exceptionally large. Indeed, performing this optimization for more than one or two layers with modest-size data sets was hardly tractable in the early days (circa 1990s) of neural networks. Modern deep learning is powered by significant advancement in computational capabilities, especially modern GPUs, and innovations in optimization algorithms. For now, we will highlight how gradient descent, which has already been covered in Chapter 5, can be used for performing the compositional curve fit (6.2.4).

6.2.1 Backpropagation (chain rule) and gradient descent

To determine the parameters $\boldsymbol{\theta}$ of the curve fit (6.2.4), an objective function must be proposed. The simplest is exactly what was used in the curve fitting chapter: root-mean square error. Thus, we propose the following error to be minimized

$$E = \sum_{j=1}^{m} \frac{1}{2}(\mathbf{y}_j - \tilde{\mathbf{y}}(\mathbf{x}_j, \boldsymbol{\theta}))^2 \tag{6.2.6}$$

where the error is computed over m data points. As an optimization problem, gradient descent allows us to minimize the error by updating the weights $\boldsymbol{\theta}$

$$\text{argmin}_{\boldsymbol{\theta}}\, E = \text{argmin}_{\boldsymbol{\theta}} \sum_{j=1}^{m} \frac{1}{2}(\mathbf{y}_j - \widetilde{\mathbf{y}}(\mathbf{x}_j, \boldsymbol{\theta}))^2. \tag{6.2.7}$$

Gradient descent updates the weights by computing the gradient of the error as a function of the parameters $\boldsymbol{\theta}$. Thus, gradient descent gives the iterative scheme

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \delta \nabla E(\boldsymbol{\theta}_k) \tag{6.2.8}$$

where δ is the *learning rate*. This formula can be verified by reviewing the gradient descent algorithm in Chapter 5. In most neural network training, the $\boldsymbol{\theta}$ are randomly seeded. And for such high-dimensional optimization problems, every time a neural network is retrained, it produces a different set of weights $\boldsymbol{\theta}$. But each of these neural networks have approximately the same performance in accuracy. This highlights the highly complex landscape of the high-dimensional loss function E. Alternatively, one can envision that there are many ways to *walk downhill* in gradient descent, given a high-dimensional landscape represented by $\boldsymbol{\theta}$.

There are two critical innovations allowing us to make (6.2.8) computationally tractable: *Backpropagation (backprop)* and *stochastic gradient descent* (SGD). Simple in conception, these two methods revolutionized and enabled the first generation of deep neural networks. Backprop solves the problem of how to compute the gradient $\nabla E(\boldsymbol{\theta}_k)$ in an efficient manner, while SGD solves the problem of how to handle large data sets and make tractable the summation over m in (6.2.6).

To begin, we consider SGD as a critically enabling part of the optimization. The SGD is a simple modification of the standard gradient descent algorithm. Instead of the computing error for all data points m, we instead randomly select a small *batch* of data for determining the gradient $\nabla E(\boldsymbol{\theta}_k)$. Suppose we select a batch size of $\tilde{m}$, where $\tilde{m} \ll m$. Then we can compute the gradient using only these randomly selected points. We can denote the update rule now as

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \delta \nabla E_K(\boldsymbol{\theta}_k) \tag{6.2.9}$$

where K represents the $\tilde{m}$ data points selected at random in a batch. At each iteration step, a new random batch is selected (previously used data points are typically excluded). A pass made through all the data during iteration constitutes an *epoch* in training. It is obvious why this saves compute time since computing the sum over $\tilde{m}$ is much faster than computing it over m when $\tilde{m} \ll m$. What is not so obvious, but is exceptionally helpful, is that the SGD with its random selection of batches can more easily progress through local minima that are common in high-dimensional landscapes. Thus, SGD gives two significant advantages. Although SGD is still commonly used, more sophisticated variants aim to further improve on the concept both in terms of computational speed and in terms of avoiding being stuck in local minima.

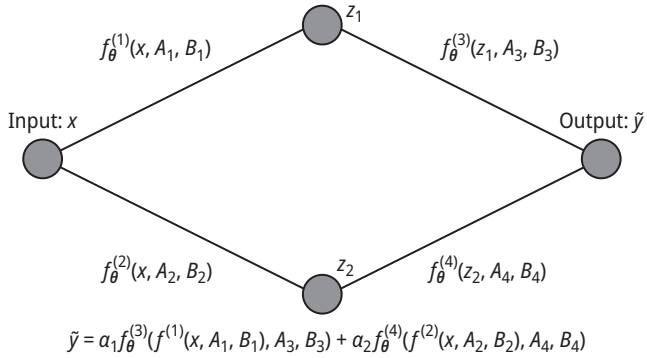


Figure 6.2 A simple feedforward neural network with a single hidden layer of two nodes. The weights and activation functions are prescribed for each connection along with the final weights, which learn how to add the hidden layer variables together. This is an exceptionally simplified version of Fig. 6.1.

The second important and enabling concept is backprop, or chain rule. When taking a derivative of the curve fit (6.2.4), the compositional structure of the model requires chain rule differentiation. So for instance, if we want to learn how to update the parameters of the first layer $\boldsymbol{\theta}_1$ through gradient descent, then we would have to use our chain rule from calculus to progressively differentiate through each layer $f^{(k)_{\boldsymbol{\theta}_k}(\cdot)}$ in order to finally produce a gradient with respect to the parameters in $\boldsymbol{\theta}_1$.

To more effectively illustrate the backprop algorithm, consider the simple model in Fig. 6.2. This is a neural network with a single hidden layer of two nodes. The figure labels all the weights and connections explicitly in order for us to show how the chain rule calculation works in practice. The curve fit for this example results in the fitting function

$$\tilde{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}) = \alpha_1 f^{(3)}(f^{(1)}(\mathbf{x}, A_1, B_1), A_3, B_3) + \alpha_2 f^{(4)}(f^{(2)}(\mathbf{x}, A_2, B_2), A_4, B_4) \tag{6.2.10}$$

where we need to update the following weights and coefficients of the model

$$\boldsymbol{\theta} = [\alpha_1, \alpha_2, A_1, A_2, A_3, A_4, B_1, B_2, B_3, B_4]\,. \tag{6.2.11}$$

In this case, there are ten total parameters that need to be determined for this curve fitting process. Thus, the gradient needs to be computed for each parameter in order to produce an update through gradient descent.

As a concrete example, consider the gradient of the error as a function of the weight A_1. The gradient for this variable is given by

$$\frac{\partial E}{\partial A_1} = (y - \tilde{y}) \frac{\partial \tilde{y}}{\partial z_1} \frac{\partial z_1}{\partial A_1} \tag{6.2.12}$$

where

$$\tilde{y} = f^{(3)}(z_1, A_3, B_3) \tag{6.2.13a}$$

$$z_1 = f^{(1)}(x, A_1, B_1). \tag{6.2.13b}$$

This explicitly shows how the chain rule updates the model by back propagating the error through the entire network structure. This also highlights why a factor of $1/2$ was added to the error term. It was for convenience since the gradient calculation brings down the power of 2 in order to cancel the factor of $1/2$.

Backprop and SGD allow for optimizing the neural network model in a tractable manner. One thing is left to do, actually compute the derivatives in (6.2.12) and (6.2.13b). Since derivatives are going to appear everywhere in the SGD, it is imperative to have simple functions to differentiate. In fact, it is highly advantageous to have functions whose derivatives are known analytically. This allows for a direct evaluation of the derivative without having to compute it. The *activation* functions $f^{(k)_{\theta_k}(\cdot)}$ used for modelling each layer are typically simple (nonlinear) functions mapping inputs to output. Commonly used activation functions are the following:

$$f(x) = x \quad - \text{ linear} \tag{6.2.14a}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \quad - \text{ binary step} \tag{6.2.14b}$$

$$f(x) = \frac{1}{1 + \exp(-x)} \quad - \text{ logistic (soft step)} \tag{6.2.14c}$$

$$f(x) = \tanh(x) \quad - \text{ TanH} \tag{6.2.14d}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad - \text{ rectified linear unit (ReLU).} \tag{6.2.14e}$$

There exist many other possibilities, but these are simple to differentiate (piecewise) analytically so that evaluation of the transfer function and its gradient can be easily computed.

The entire coding infrastructure for building neural networks, including gradient computations for a selected activation function, is now readily available with pyTorch, JAX, MATLAB, or Julia. We can all feel lucky as a community that we do not have to write all our own code from scratch that would accommodate all the hyper-parameter tuning we may want to do with neural networks. In the feedforward network illustrated here, the hyper-parameters include the number of hidden layers, the size of each hidden layer, the activation functions, the batch size, the learning rate, the optimization routine to use, and the objective (loss) function to use for training. Hyper-parameter tuning is typically the most time-intensive portion of the curve fitting process, as good results typically depend critically on being in a good hyper-parameter regime.

6.3 Learned Models: Generalization, Interpolation, and Extrapolation

Curve fitting with neural networks will be shown on a simple and intuitive example. What will be compared here is a neural network approximation to more traditional curve fitting methods, like lines or simple functions. This example allows for the evaluation of generalization of the model, specifically through interpolation or extrapolation. Consider the *true curve*:

$$y = a\cos(bx) + cx + d \tag{6.3.1}$$

where the parameters $a, b, c,$ and d are prescribed. The data for this curve will include Gaussian distributed white noise so that our *noisy data* is given by

$$y_n = y + \mathcal{N}(0, \sigma) \tag{6.3.2}$$

where σ parametrizes the strength of the noise. The goal is simple, use curve fitting to build a model that fits the data and extracts the true solution as best as possible. Further, evaluate the model on withheld data that is interpolatory or extrapolatory in nature.

Figure 6.3 shows the two situations we will consider. In panel (a), the noisy training data is selected from the range $x \in [20, 40]$. The goal will be to build a curve fit model from this data and predict what happens in the extrapolatory regime $x \in [0, 20]$ and $x \in [40, 50]$. In panel (b), the noisy training is selected from the first third and last third of the data on the interval of length L, thus the training come from $x \in [0, L/3]$ and $x \in [2L/3, L]$. The test region is an interpolatory regime $x \in [L/3, 2L/3]$. The goal is then to see how some of the different curve fitting paradigms work on this fairly simple noisy data set. Despite its simplicity, one will see that it can be quite difficult to fit even such simple curves with a neural network.

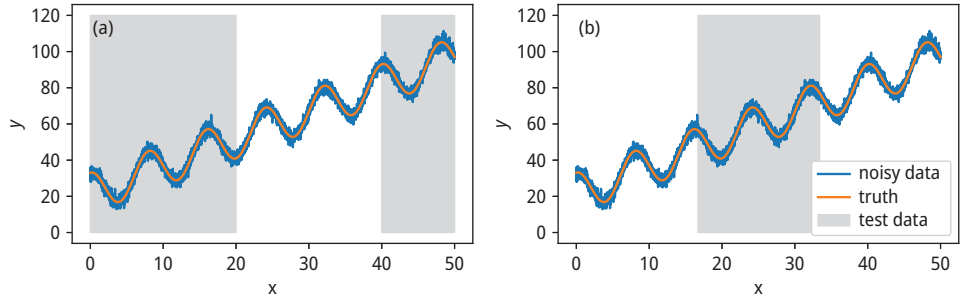


Figure 6.3 A simple curve fitting example where the goal is to use the noisy training data (blue) to build a model of the underlying truth (orange). In panel (a), the training data is selected from the range $x \in [20, 40]$ with the goal of extrapolating with the model the left and right of this domain. In panel (b), the first and last third of the data is used for training with the goal of modelling the middle third (interpolation) of the data.

The data is made with the following code block, where the parameters of the underlying curve are specified. Noise is added to make the data that is used for curve fitting.

```
x = np.arange(0,50.01, 0.01)
a = 11; b = np.pi/4; c = 1.5; d = 22
y = a*np.cos(b*x) + c* x + d

n = len(y); noise = np.random.randn(n)
yn = y+2*noise
```

The strength of the noise term, which is chosen to be two in the above example, can be easily modified to make this curve fitting even more difficult.

Before fitting a neural network, consider the more traditional methods of fitting. Specifically, we can try fitting the data with the following two models:

$$f(x) = Ax + B \tag{6.3.3a}$$

$$f(x) = A\cos Bx + Cx + D \tag{6.3.3b}$$

where the parameters A, B, C, and D are determined by least-square fitting. For a line fit (6.3.3b), the method of the curve fitting chapter shows that determining A and B is accomplished by solving a 2×2 system of equations. The curve (6.3.3b) is more difficult to achieve, as it involves solving a 4×4 nonlinear system of equations as shown in Section 4.3. The fit of the two curves are generated with the following:

```
pcoeff = np.polyfit(xt, yn, 1)
yp = np.polyval(pcoeff, x)

def cosinefit(c, x, y):
    e = np.sum((c[0]*np.cos(c[1]*x)+c[2]*x+c[3]-y)**2)
    return e

c0 = np.array([10, 1*np.pi/4, 5/3, 20])
res= opt.minimize(cosinefit, c0, args=(xt, yn), method='Nelder-Mead')
   c = res.x
yfit2 = (c[0]*np.cos(c[1]*x)+c[2]*x+c[3])
```

The results of these fits are shown in Fig. 6.4. Specifically, Fig. 6.4(a) shows the results given the noisy training data on the interval $x \in [20, 40]$. The line fit (6.3.3b) and sinusoidal fit (6.3.3b) are both shown to capture the general behaviour of the data well outside the training interval. Indeed, the sinusoidal fit (6.3.3b) is nearly a perfect fit since we actually

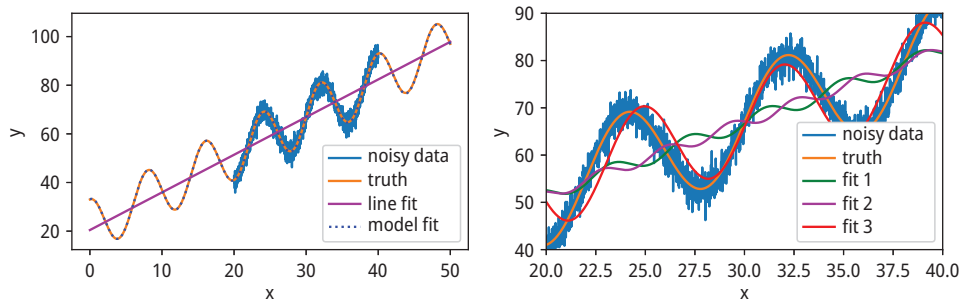


Figure 6.4 Given training data in the range $x \in [20, 40]$, a line fit (6.3.3b) and a sinusoidal fit (6.3.3b) are used to approximate the data. (a) The line fit and sinusoidal fit results are shown, where a good guess $[A, B, C, D] \approx [10, \pi/4, 5/3, 20]$ is given for the sinusoidal fit. The sinusoidal fit is nearly perfect, and both fits extrapolate well outside of the noisy training data. (b) The sinusoidal fit given three different guesses for the parameters $[A, B, C, D]$. A poor starting point for solving the nonlinear system of equations shows that local minima are reached with poor fitting results. A good initialization of the parameters is certainly required for nonlinear curve fits.

picked a function form consistent with the true data. However, the sinusoidal fit results in a 4×4 nonlinear system of equations, which means there can be multiple solutions, or multiple minima, to the optimization problem. Nonlinear systems are typically solved via an iteration procedure like gradient descent (in this case a Nelder–Mead iteration scheme), and so the initial starting point of the iteration is important. Figure 6.4(b) shows the same sinusoidal curve fitting procedure as in panel (a), but with different initial guesses for the iteration starting point. This shows how different the fitting can be simply by changing the initialization of the iteration procedure. The three curves show that the performance of nonlinear curve fitting is highly sensitive. The reason for illustrating this is because neural networks have the same issue: the network weights are typically all randomly seeded at the beginning of training. And unlike the sinusoidal curve, where a good starting point can be guessed, a good start point for neural networks is typically impossible to construct.

The goal is now to use a neural network to do the curve fitting. The neural network is a nonlinear, compositional function fit to the data. As already illustrated in the sinusoidal fit (6.3.3b), nonlinear curve fitting is difficult due to the landscape of local minima that can be present. This is especially true for neural networks, which have an exceptionally high-dimensional number of parameters that require determining. SGD, and many other innovations since, help neural networks not get stuck in local minima, but this remains an important issue. In what follows, a neural network will map the x-values of the data to the y-values of the data. The data has already been illustrated in Fig. 6.3 for the two cases of interest.

There are options for building neural networks, including pyTorch, JAX, Julia, and MATLAB. Torch will be the tool used here and throughout the book, as it is the most commonly used across the community. Torch can be easily installed with a pip install. Once installed, the following lines of code import torch, scale the data, and then transform the data to a torch data format.

```
import torch
from torch.utils.data import DataLoader
import sklearn.preprocessing as skp

sc = skp.MinMaxScaler()
xt = x[2000:4000].reshape(-1,1)
yn = ynoise[2000:4000].reshape(-1,1)
xt = sc.fit_transform(xt)
sc2 = skp.MinMaxScaler()
yt = sc2.fit_transform(yn)
xt = torch.tensor(xt).to(torch.float32)
yt = torch.tensor(yt).to(torch.float32)
```

Note that scaling data is critically important for successful model-building with deep neural networks. In this case, the training data is selected from the range $x \in [20, 40]$ and scaled with the commonly used MinMaxScaler function. This is just one of the many possibilities for scaling the data. Thus, one can already see the challenges of using neural networks, many choices are required to be made. This broadly falls under *hyper-parameter* tuning, whereby the various aspects of architecture, scaling, optimization, activations, layers, size, etc., must all be chosen. Unfortunately, these choices have significant impact on the performance of the network. So hyper-parameter tuning is an integral and time-consuming part of making deep learning work.

The next module in the curve fitting is to construct a model, which explicitly knows that the curve fit data is *sequential*. That is, the x-values and y-values in the curve are not just scattered, but rather come as a sequence as one advances from smaller x-values to larger x-values. The following sets up the structure for the data.

```
class TimeSeriesDataset(torch.utils.data.Dataset):
    '''Takes input sequence of sensor measurements with
    shape (batch size, lags, num_sensors)
    and subsequent measurement, return Torch dataset'''
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
        self.len = X.shape[0]

    def __getitem__(self, index):
        return self.X[index], self.Y[index]

    def __len__(self):
        return self.len
train_dataset = TimeSeriesDataset(xt,yt)
```

The next component is to specify the neural network used. Here, we will build a feedforward neural network (FNN). The following allows for building a customizable FNN, which specifies the input size of the data, output size of the data, the total number of layers, the size of the hidden layers, and the activation functions to use.

```
class CustomFNN(torch.nn.Module):
    def __init__(self,input_size,hidden_size,output_size,
         num_layers,activation):
        super(CustomFNN, self).__init__()
        self.layers = torch.nn.ModuleList()
        self.layers.append(torch.nn.Linear(input_size, hidden_size))
        for _ in range(num_layers - 1):
            self.layers.append(torch.nn.Linear(hidden_size,
                hidden_size))
        self.layers.append(torch.nn.Linear(hidden_size, output_size))

        if activation == 'relu':
            self.activation = torch.nn.ReLU()
        elif activation == 'elu':
            self.activation = torch.nn.ELU()
        else:
            raise ValueError("Invalid activation function")

    def forward(self, z):
        for layer in self.layers[:-1]:
            z = layer(z)
            z = self.activation(z)
        z = self.layers[-1](z)
        return z
```

In the example we consider, the input size is the one, which is the x-values of the data, and the output size of the data is one, which is the y-values of the data. Here one can choose between relu or elu activation functions. This code specifies the neural network architecture, or curve, that is fitting the data.

Finally, we train the neural network specified. Training simply means using gradient decent, or any other optimization technique, to find the parameters of the curve fit, i.e. the weights of the neural networks. In this case, the popular ADAM optimizer is used with a specified learning rate and for a specified number of epochs. Note that the neural network is now specified with three hidden layers of size 30 and activation functions elu.

```
model = CustomFNN(1, 30, 1, 3, 'elu')
train_loader = DataLoader(train_dataset, batch_size=64)

lr = 1e-3; epochs = 5000
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
criterion = torch.nn.MSELoss()
for epoch in range(epochs):

    for _, data in enumerate(train_loader):
        model.train()

        outputs = model(data[0])
        optimizer.zero_grad()

        loss = criterion(outputs, data[1])
        loss.backward()
        optimizer.step()

    if epoch % 100 == 0:
        print(epoch)
        print(loss.item())
```

This training function updates the weights of the neural network in an iterative process. The number of parameters that can be used for hyper-parameter turning are quite large, including the number of hidden layers used, the size of the hidden layers, the activation functions used, the learning rate, the optimizer, etc. The code trains the model until a specified level of performance is achieved, or the total number of epochs has been performed. The data is then unscaled and plotted using the following.

```
transformed_x_test = sc.transform(x.reshape(-1,1))
test_out = model(torch.tensor(transformed_x_test).to(torch.float32))

ytemp = test_out[:,0].detach().numpy()
y1extra = sc2.inverse_transform(ytemp.reshape(-1,1))

plt.plot(xorig,y1extra)
plt.plot(xorig,yorig)
```

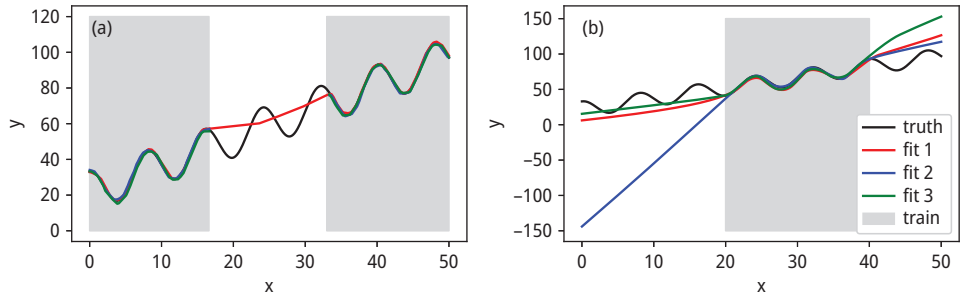


Figure 6.5 Curve fitting capabilities of a feedforward neural network with three hidden layers of size 30 each and elu activation functions. Three trained models are shown, each with different random initializations. (a) In the interpolatory setting, the neural network shows strong capabilities for fitting over the training regime. In the interpolation regime, the network approximates the middle third of the data with a simple curve and misses the oscillatory behaviour. (b) In the extrapolatory setting, the neural network again fits over the training regime quite well. But in the extrapolatory regime, the model can be quite off. As a general rule, neural networks with enough data will fit exceptionally well over regimes where data is collected. But neural networks in general are exceptionally poor at extrapolation.

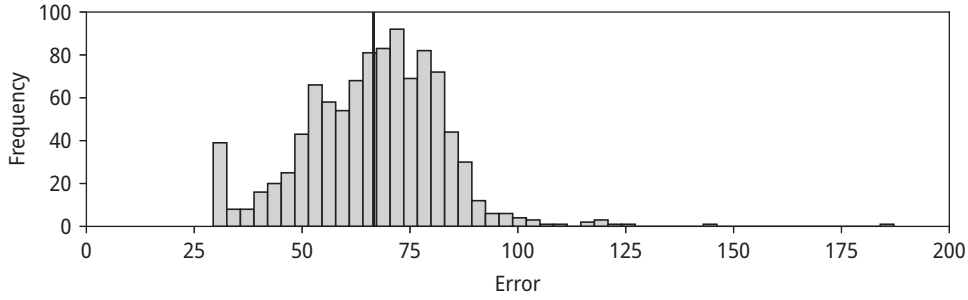


Figure 6.6 Distribution of errors for fitting the data from random initial starts of the neural network. Note that for this application, where the test set is in an extrapolation regime, the error distribution is significant, ranging from a minimal RMSE of approximately 29 to a value of 185. The average error is about 67. This highlights an important aspect of neural network training: the performance on a true withheld test set can have very high variance.

The test data set is compared against the approximation to the test data. The performance on the test data is the most important part of the comparison. Figure 6.5 shows the performance of the neural network on the interpolatory and extrapolatory data sets. Three different neural networks are trained, simply by re-running the code. Recall that at each running of the code, the neural network is randomly seeded. Thus at each run, different local minima are obtained in the loss landscape. Importantly, one can see from both Figs. 6.5(a) and (b) that the neural network fits the training data exceptionally well. However, when generalization is required, for the interpolatory regime of Fig. 6.5(a) or the extrapolation regime of Fig. 6.5(b), the neural network suffers in performance. This is highlighted further in Fig. 6.6, where the

distribution of errors over 1000 training runs is demonstrated. The variance on the test set is quite large, illustrating one of the important aspects of neural network training for extrapolation problems. For the interpolation regime, the behaviour is not terrible, but the neural network never learns the oscillatory behaviour of the training data. For the extrapolation regime, the neural network is unconstrained and the behaviour can be quite poor. One should keep these results in mind. Neural networks are not magic, they are ultimately curve fits that perform well on interpolation, but are highly suspect in extrapolation. The extraordinary success of machine learning in speech, vision, and language illustrate the fact that we have enough data in these domains to empower the deep learning to become the ultimate interpolation engine for various applications in these domains.

7 Visualization

7.1 Customizing Plots and Basic 2D Plotting 152
7.1.1 Basic 2D plotting 155
7.2 More 2D and 3D Plotting 160
7.2.1 3D plots 162
7.3 Movies and Animations 165
7.3.1 GIF animation files 167

Visualization is one of the most important aspects of Python that needs to be considered when communicating your results with others. The default settings on Python are often lacking when professional-looking graphs and plots need to be produced. In the following subsections, both full customization of graphs as well as the high-end two- and three-dimensional graphic capabilities of Python are considered. The final subsection will then consider how to use these plotting algorithms to produce movie and animation files.

7.1 Customizing Plots and Basic 2D Plotting

Section 1.5 illustrates many of the basic features of the plotting algorithms of Python. However, it is rarely the case that a plot created with the default line thicknesses and font sizes of Python can be used for professional purposes. Thus, customization of the plot is important to do. Further, if one wishes to use the same custom setting for a large number of plots, it is much better to have a customized Python code that can be applied to all the data sets. All the figures in this book have been customized in order to get the font sizes and line thicknesses correct. Indeed, the default line thicknesses and text font sizes typically show up poorly when printed out in a report or manuscript unless they are modified.

To begin the consideration of the customization process, the following specific functions will be plotted:

$$f(x) = \cos(x) \tag{7.1.1}$$

$$g(x) = \sin(0.2x^2)\exp(-0.02x^2) \tag{7.1.2}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0008

on the interval $x \in [-10, 10]$. These two functions can easily be made by using the Python commands

```
L = 10
x = np.linspace(-L, L, 100)
f = np.cos(x)
g = np.sin(0.2 * x**2) * np.exp(-0.02 * x**2)
```

Note that in this example, the functions have been discretized into 100 points.

Plotting these functions can be done trivially in Python with the commands

```
plt.subplot(2, 1, 1)
plt.plot(x, f, 'm', label='f(x)')
plt.plot(x, g, 'k', label='g(x)')
plt.legend()
plt.title('$\\xi^2$ dependence', fontsize=12)
```

The top panel of Fig. 7.1 shows the default image generated from Python. Although a subjective opinion, the default font size and line thickness are simply not appropriately chosen for the purpose of insertion into a manuscript or report.

To customize the plot, access to its various properties must be obtained. The line width can easily be adjusted with the command structure:

```
plt.subplot(2, 1, 2)
plt.plot(x, f, 'm', linewidth=6)
plt.plot(x, g, 'k', linewidth=2)
```

This sets the thickness to a prescribed value. Generally, it is best to have a line thickness of a least value 2. The default value is unity.

The following lines of code set the axis limits, where tick marks are to be displayed, and what is to be displayed at each tick mark. Further, the font size is also set.

```
plt.xlim([-10, 10])
plt.xticks([-10, -7.5, -5, -2.5, 0, 2.5, 5, 7.5, 10],
      ['-L', '', '-5', '', '0', '', '5', '', 'L'])
plt.yticks([-1.5, -1, -0.5, 0, 0.5, 1.0, 1.5],
     ['', 'min', '', '0', '', 'max', ''])
plt.xlabel('space $\\xi$', fontsize=12)
```

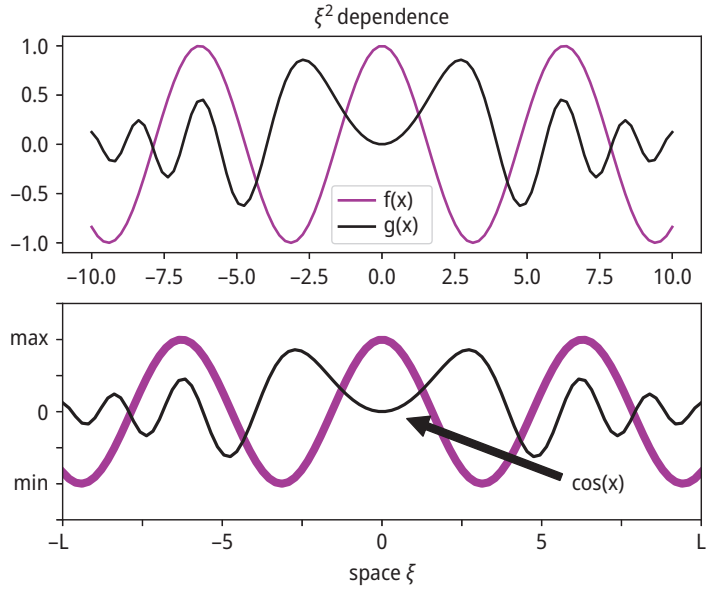


Figure 7.1 Plots of the functions $f(x) = \cos(x)$ and $g(x) = \sin(0.2x^2)\exp(-0.02x^2)$ over the interval $x \in [-10, 10]$. The top graph is plotted with the default settings of Python. The bottom graph is a fully customized plot with font size and line thickness adjusted to more appropriate settings.

The results of this modification can be seen in the bottom panel of Fig. 7.1. Thus, full control of the graph can be established and all the defaults can be overridden.

In addition to customizing the font sizes, ticks and line thicknesses, Latex commands can be used in conjunction with Python. Latex is a professional typesetting software package that is freely available. Further, it has become the dominant typesetting software in the scientific community. Most Latex commands are initiated with the backslash command. The following lines of codes show the implementation of Latex commands in the label, title, and legend. The ability to implement Latex structures in a plot is instrumental for many scientifically oriented plots, where Greek characters typically represent the quantities of interest.

In addition to customizing the properties of the graph, annotations (arrows) can be added to a plot. An arrow is specified by defining its starting (x, y) location and ending (x, y) location. The entire figure space is defined with $(0, 0)$ as the bottom left corner and $(1, 1)$ as the top right corner. All annotation locations are defined with these reference points in mind. The following command annotates one of the plot lines for the bottom graph of Fig. 7.1.

```
plt.annotate('cos(x)',xy=(0,0),xytext=(0.75,-0.45),
    textcoords='axes fraction',
        arrowprops=dict(facecolor='black', shrink=0.05), fontsize=12)
```

It should be noted that the string specified by the annotation does not handle Latex input like the axis label, legend, and title commands.

7.1.1 Basic 2D plotting

In addition to modifying and adjusting all the default plotting parameters, there is an obvious desire to consider plotting higher-dimensional functions. Our starting point is a function of two spatial dimensions x and y. For instance, we can consider the spiral wave defined as the following:

$$u(x,y) = \tanh\left[\sqrt{x^2+y^2}\cos\left(m\angle(x+iy) - \sqrt{x^2+y.^2}\right)\right] \tag{7.1.3}$$

where the $\angle$ denotes the phase angle of the quantity $(x+iy)$. In Python, such a function can be defined as the following:

```
L = 10; x = np.linspace(-L, L, 50); y = x
X, Y = np.meshgrid(x, y)
m = 1  # number of spirals
u = np.tanh(np.sqrt(X**2 + Y**2)) * np.cos(m * np.angle(X
        + 1j * Y) - np.sqrt(X**2 + Y**2))
```

Here the parameter m is the number of spiral arms in the function. There are a variety of techniques for producing a graphical representation of this 2D function. Figure 7.2 displays the first set of visualizations of the spiral wave. The four panels are given by the Python code:

```
fig = plt.figure(); fig = plt.figure(figsize=(18, 18))

ax = fig.add_subplot(221, projection='3d')
ax.plot_surface(X, Y, u, cmap='coolwarm')
plt.gca().set_xticks([]); plt.gca().set_yticks([]); plt.gca().
    set_zticks([])

ax = fig.add_subplot(222, projection='3d')
ax.plot_surface(X, Y, u, cmap='coolwarm')
ax.contour(X,Y,u,zdir='u',offset=np.min(u),cmap='coolwarm',
    linestyles='solid')

ax = fig.add_subplot(223, projection='3d')
ax.contourf(X, Y, u, cmap=cm.coolwarm)

ax = fig.add_subplot(224, projection='3d')
ax.plot$\_$wireframe(X, Y, u, color='black', alpha=0.5)
```

The ticks and labels have all been removed for visualization purposes. The **surface** command is the standard surface plot and is displayed in the top left of Fig. 7.2. A surface plot and a contour plot can be combined (top right of Fig. 7.2). The **contourf** command is a combined contour and surface (bottom left of Fig. 7.2), while the **plot_wireframe** command is simply the mesh of the data (bottom right of Fig. 7.2).

Other options exist for plotting the data. One of the most important is the **pcolor** command which plots a colour scale projection of the data. This topographical plot is often a very insightful way to visualize the data, especially if it has a complicated structure which can be hidden in one of the other plotting options of Fig. 7.2. The following plotting options produce the topographical visualization:

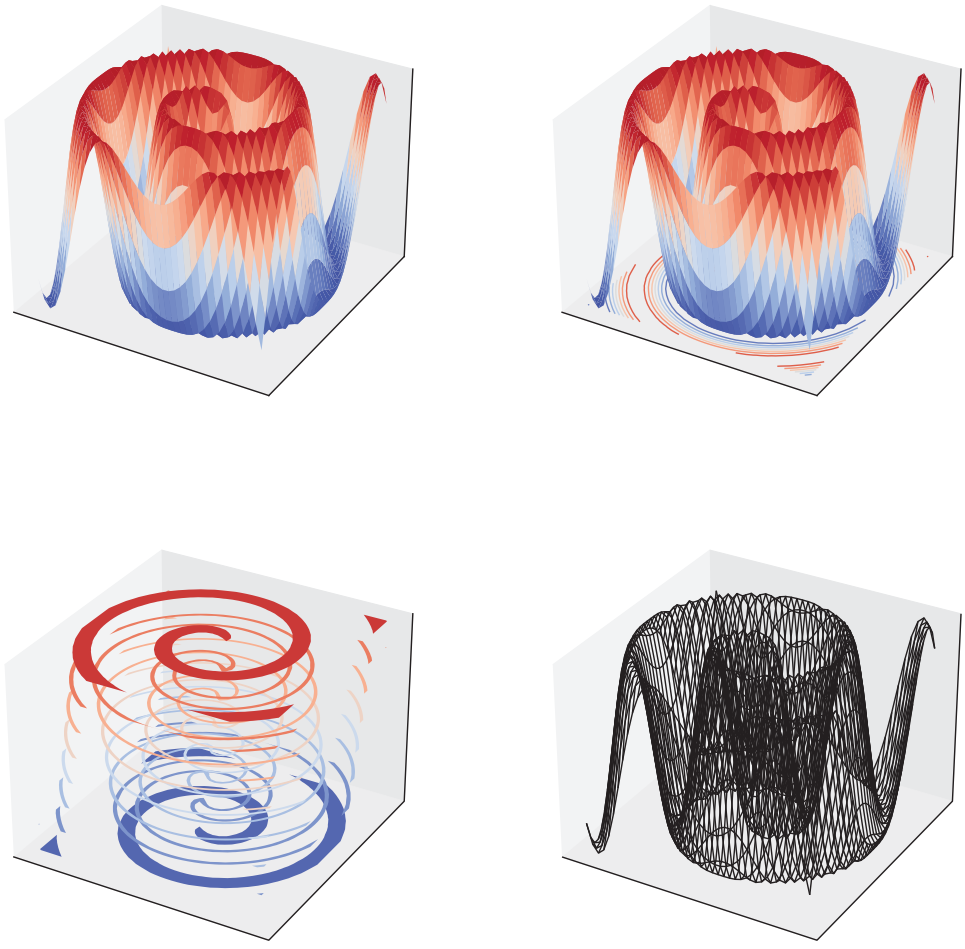


Figure 7.2 Some of the common options for plotting of data including **surface** (surface plot), surface plot with a contour, combination of surface and contour, and **wireframe** (just the mesh of the data).

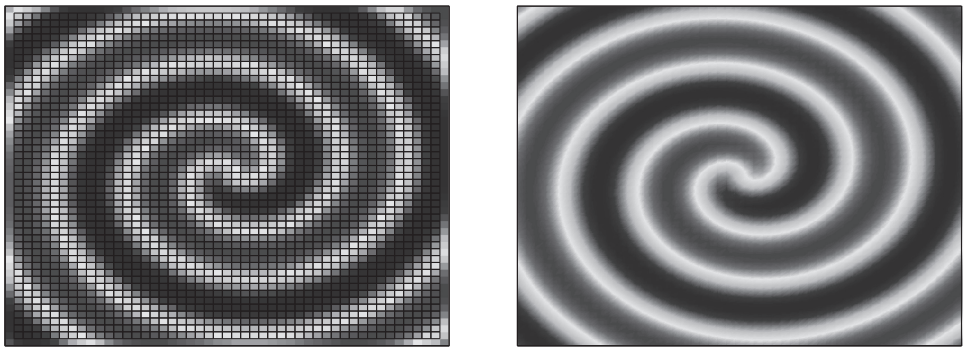


Figure 7.3 Topographical plot of the spiral wave. Note that the **shading interp** command smooths out the data by interpolation (right panel).

```
ax = fig.add_subplot(221)
ax.pcolor(x, y, u, edgecolors='k', linewidths=1, cmap='coolwarm')
ax = fig.add_subplot(222)
ax.pcolor(x, y, u, cmap='coolwarm')
```

Note that the **shading interp** command interpolates over the matrix values given by $\mathbf{u}$. This is often a nice option for visualization. Figure 7.3 shows the results of the **pcolor**.

Different colour schemes can also be considered in the plotting. In Fig. 7.4, a variety of colour schemes are considered with the **surface** command. The following code generates this figure:

```
fig = plt.figure(figsize=(18, 18))
ax = fig.add_subplot(221, projection='3d')
ax.plot_surface(X, Y, u, color='red')
plt.gca().set_xticks([]); plt.gca().set_yticks([]);
    plt.gca().set_zticks([])

ax = fig.add_subplot(222, projection='3d')
ax.plot_surface(X, Y, u, cmap='hot')

ax = fig.add_subplot(223, projection='3d')
ax.plot_surface(X, Y, u, cmap='gray')

ax = fig.add_subplot(224, projection='3d')
ax.plot_surface(X, Y, u, color='gray')
```

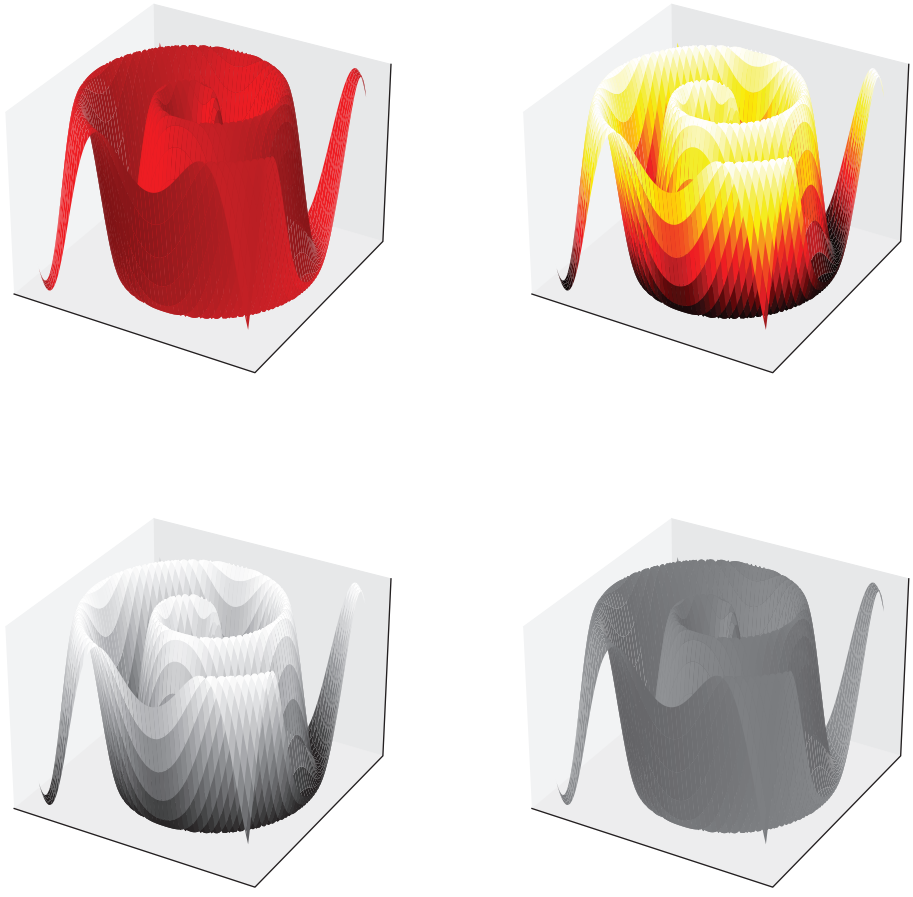


Figure 7.4 Lighted surface plots in differing colour maps and greyscale.

Note that each of the figures generated are different figure numbers. This is due to the fact that only one colormap can be implemented per figure.

Finally, the **plot_wireframe** command is considered for plotting. In Fig. 7.5, the wireframe command is used with a black-and-white colormap. The two plots generated in this figure show some nice options for grey-scale plotting. Often for journals, black-and-white is an important option due to the cost of printing in colour. The code for this is the following:

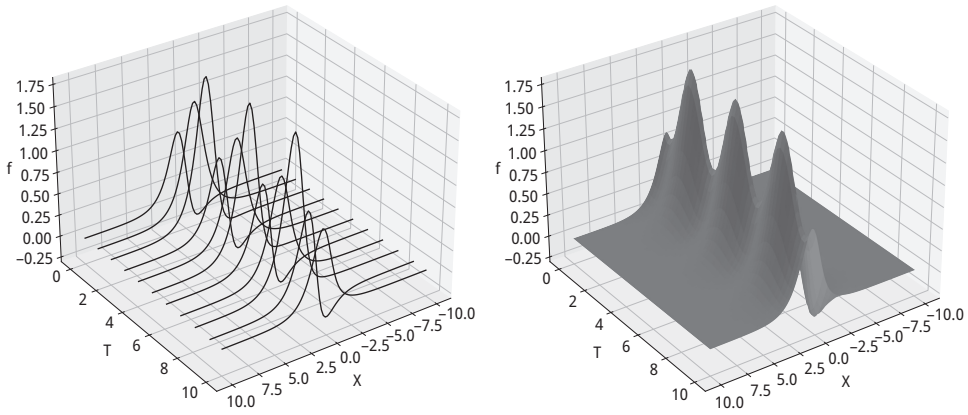


Figure 7.5 Wireframe and lighted surface plots in black and white and greyscale.

```
x = np.linspace(-10, 10, 100)
t = np.linspace(0, 10, 50)
X, T = np.meshgrid(x, t)
f = np.cosh(X)**(-1) * (1 - 0.5 * np.cos(2 * T))
+ (np.cosh(X)**(-1) * np.tanh(X)) * (1 - 0.5 * np.sin(2 * T))

fig = plt.figure()
fig = plt.figure(figsize=(18, 18))

ax = fig.add_subplot(221, projection='3d')
ax.plot$\_$wireframe(X, T, f, color='k', rstride=5, cstride=0)
ax.set_xlabel('X'); ax.set_ylabel('T'); ax.set_zlabel('f')
ax.view_init(elev=30, azim=55)

ax = fig.add_subplot(222, projection='3d')
surf = ax.plot_surface(X, T, f, color='gray', antialiased=False)
ax.set_xlabel('X'); ax.set_ylabel('T'); ax.set_zlabel('f')
ax.view_init(elev=30, azim=55)
```

The **plot_wireframe** command is especially useful for showing spatio-temporal dynamics of a system in a simple way and in black-and-white.

7.2 More 2D and 3D Plotting

The aim of this section will be to outline some of the plot combining methods that can be used for 2D functions, as well as develop visualization methods for 3D functions. 3D functions are particularly difficult because all of the spatial dimensions x, y, and z already fill up the visualization domain. Thus, methods must be available that allow for essentially 4D data, i.e. the value of some function $u(x, y, z)$ expressed as a function of all three spatial dimensions.

To begin, 2D combination plots are considered. Specifically, the spiral wave of the last section is revisited. Thus, the function to be plotted is generated from the code:

```
L = 10; x = np.linspace(-L, L, 50); y = x
X, Y = np.meshgrid(x, y)

m = 1  # number of spirals
u = np.tanh(np.sqrt(X**2 + Y**2)) * np.cos(m * np.angle(X
                        + 1j * Y) - np.sqrt(X**2 + Y**2))
```

As before, only a single spiral wave will be considered as a function of the spatial variables x and y.

A combination plot can be extremely helpful in visualizing a solution. Specifically, when considering the visualization of the spiral in the previous section, the spiral was well represented by both the **meshgrid** (lighted surface) and **pcolor** (topographical) plots. However, each of these plots has drawbacks. In the case of the lighted surface, portions of the spiral wave block out what is behind it. The topographical plot generated from **pcolor**, on the

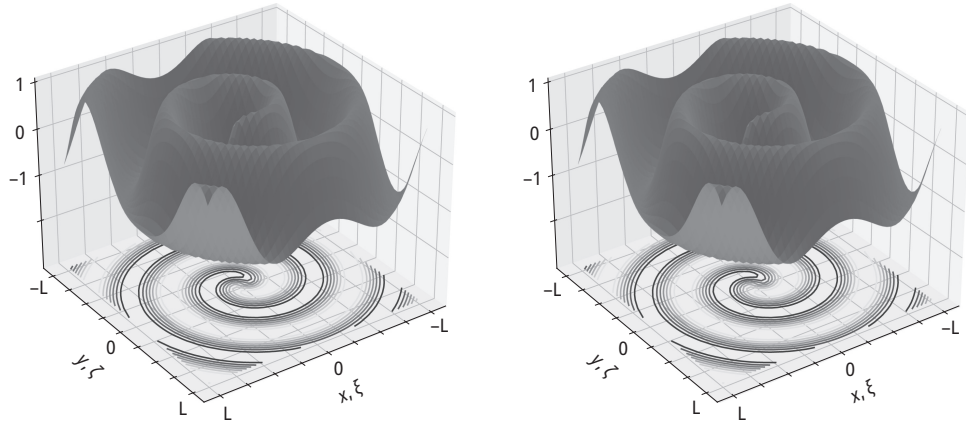


Figure 7.6 Combination plot of the spiral wave using both the **meshgrid** and **pcolor** plotting commands. The z-axes have been relabelled in order to accurately reflect the z values. Further, the right figure has been made semi-transparent by setting **alpha(0.8)**.

other hand, gives a complete representation of the solution without blocking anything out. But in this case, it is difficult to see what the heights correspond to in terms of units. One can certainly add a **colorbar** to the plot which will map the colours to physical values, making it a more user-friendly plot. Alternatively, one could combine both the **meshgrid** and **pcolor** together to give an excellent representation of the solution.

For combination plots, the **hold on** command must be used in order to keep the various plot commands from overwriting the previous plot. Here, the aim will be to plot both a **meshgrid** and **pcolor** figure together in combination. Figure 7.6 depicts the net effect of this process. The code for doing this is as follows:

```
ax = fig.add_subplot(221, projection='3d')
ax.plot_surface(X, Y, u + 2, color='gray')
ax.contour(X,Y,u,zdir='u',offset=np.min(u),cmap='gray',
   linestyles='solid')

plt.xlabel(r'$x, \xi$', fontsize=15, fontweight='bold')
plt.ylabel(r'$y, \zeta$', fontsize=15, fontweight='bold')
plt.xticks([-L, -7.5, -5, -2.5, 0, 2.5, 5, 7.5, L],
   ['-L', ' ', ' ', ' ', '0', ' ', ' ', ' ', 'L'], fontsize=15)
plt.yticks([-L, -7.5, -5, -2.5, 0, 2.5, 5, 7.5, L],
   ['-L', ' ', ' ', ' ', '0', ' ', ' ', ' ', 'L'], fontsize=15)
ax.set_zticks([0, 1, 2, 3])
ax.set_zticklabels(['', '-1', '0', '1'], fontsize=15)
ax.view_init(elev=30, azim=55)

ax = fig.add_subplot(222, projection='3d')
ax.plot_surface(X, Y, u + 2, color='gray', alpha=0.7)
ax.contour(X,Y,u,zdir='u',offset=np.min(u),cmap='gray',
   linestyles='solid')

plt.xlabel(r'$x, \xi$', fontsize=15, fontweight='bold')
plt.ylabel(r'$y, \zeta$', fontsize=15, fontweight='bold')
plt.xticks([-L, -7.5, -5, -2.5, 0, 2.5, 5, 7.5, L],
   ['-L', ' ', ' ', ' ', '0', ' ', ' ', ' ', 'L'], fontsize=15)
plt.yticks([-L, -7.5, -5, -2.5, 0, 2.5, 5, 7.5, L],
   ['-L', ' ', ' ', ' ', '0', ' ', ' ', ' ', 'L'], fontsize=15)
ax.set_zticks([0, 1, 2, 3])
ax.set_zticklabels(['', '-1', '0', '1'], fontsize=15)
ax.view_init(elev=30, azim=55)
```

There are three key features of this plot. First, when plotting, the command

```
meshgrid(x,y,u+2)
```

is used to depict the **meshgrid**. The reason that the solution is lifted up by 2 units is that the **pcolor** only plots the solution at $z = 0$. Thus, the **color** plot would slice directly through the **meshgrid** plot and create a visually unappealing plot. Instead, the solution is lifted up by 2 so that the **pcolor** plot no longer slices through the **meshgrid** plot. This creates a problem, however, since the z-scale now shows the data goes from $z = 1$ to $z = 3$ instead of the correct $z = -1$ to $z = 1$. This can be easily fixed by overwriting the axis labels so that $z = 1$ ($z = 3$) is labelled as $z = -1$ ($z = 1$).

A second interesting feature of these plots is that the **meshgrid** plot can be made semi-transparent. In the left panel of Fig. 7.6, the plot is not transparent. However, in the right panel, the figure is made semi-transparent with the parameter **alpha**. The value of alpha can be chosen from 0 (fully transparent or invisible) to 1 (nontransparent). Here, the value was chosen to be 0.8 allowing for partial transparency. The transparency can often be used for favourable viewing of an image. As a final note, the **fontweight** command has been used to bold the axis labels.

7.2.1 3D plots

Three-dimensional plots are difficult to produce due to the fact that all three spatial dimensions are used. Thus, representing the value of the solution must be done with some other method. The simple function to be considered here is a 3D periodic function

$$u(x, y, z) = \cos(x)\cos(y)\cos(z)\,. \tag{7.2.1}$$

This is an *egg-carton* type structure in 3D space. The following lines of Python can be used to produce the function itself.

```
x = np.linspace(-3, 3, 20); y = x; z = x
X, Y, Z = np.meshgrid(x, y, z)
u = np.cos(X) * np.cos(Y) * np.cos(Z)
```

The **meshgrid** command can be easily generalized to 3D (or even higher), thus producing 3D matrices $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, and $\mathbf{U}$. For each component of the matrix $U(i, j, k)$, a value is assigned based upon the function chosen.

Python's 3D plotting capabilities are not very well advanced at this. point. As a result, what is illustrated is what can be done with MATLAB. Two nice options are available for plotting this function, the **isosurface** command and the **slice** command. The **isosurface** command uses

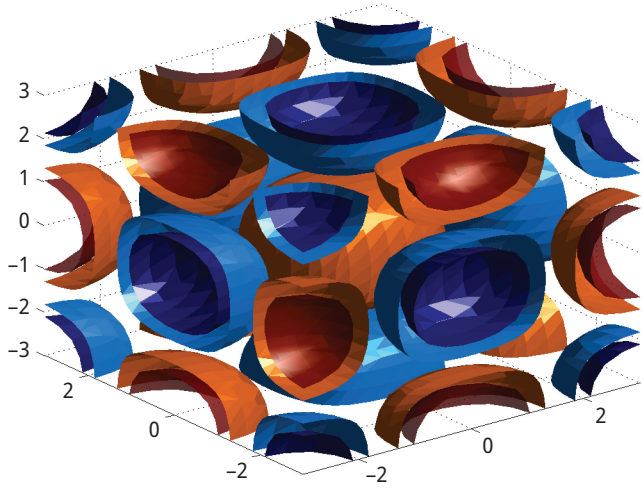


Figure 7.7 Isosurfaces generated from the **isosurface** command. Four surfaces are generated with values of –0.5, –0.25, 0.25, and 0.5. The plot has been made semi-transparent by setting alpha to 0.8.

predefined isosurface values in order to plot surfaces of constant value. As with the combination plots, the **hold on** command must be used. Figure 7.7 demonstrates the implementation of the **isosurface**. The following lines of Python are used to generate this plot:

```
isosurface(x,y,z,u,0.5), grid on
alpha(0.8), hold on
isosurface(x,y,z,u,-0.5)
isosurface(x,y,z,u,0.25)
isosurface(x,y,z,u,-0.25)
set(gca,'Xlim',[-3 3],'Ylim',[-3 3],'Zlim',[-3 3],'Fontsize',[18])
```

The isosurface values chosen in this case are for the surface height of 0.5, 0.25, −0.25, and −0.5. Thus, four isosurfaces are represented in the plot with the colours being related to the isosurface height.

As with the combination plots, two or more plot types can be incorporated together into one figure. Here, the **isosurface** is combined with the **slice** command to produce a composite figure. The **slice** command allows the user to specify **pcolor** planes on which the data can be projected. Thus, it is really a generalization of the **pcolor** plotting routine for 3D. In the

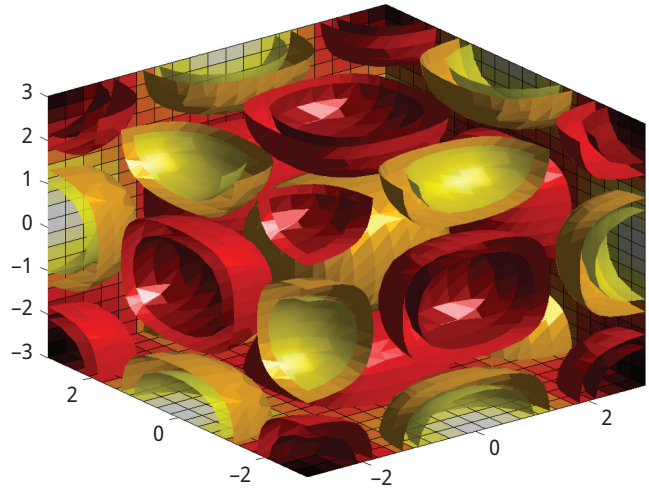


Figure 7.8 Isosurfaces generated from the **isosurface** command in combination with the **slice** command for projecting the solution on the surfaces at the edge of the domain. Four surfaces are generated with values of –0.5, –0.25, 0.25, and 0.5. The plot has been made semi-transparent by setting alpha to 0.8.

simple implementation here, slices are placed at the planes $x = 3$, $y = 3$, and $z = -3$. The following lines of code have been used to plot this.

```
isosurface(x,y,z,u,0.5), grid on
alpha(0.8), hold on
isosurface(x,y,z,u,-0.5)
isosurface(x,y,z,u,0.25)
isosurface(x,y,z,u,-0.25)
slice(x,y,z,u,3,3,-3)
set(gca,'Xlim',[-3 3],'Ylim',[-3 3],'Zlim',[-3 3], 'Fontsize',[18])
colormap(hot)
```

In this case, the hot colormap has been used in Fig. 7.8 for the visualization. This gives a combination representation, which in some visualizations can be extremely useful.

As a final example, the **slice** command is used exclusively for the 3D visualization. The following lines of code are used.

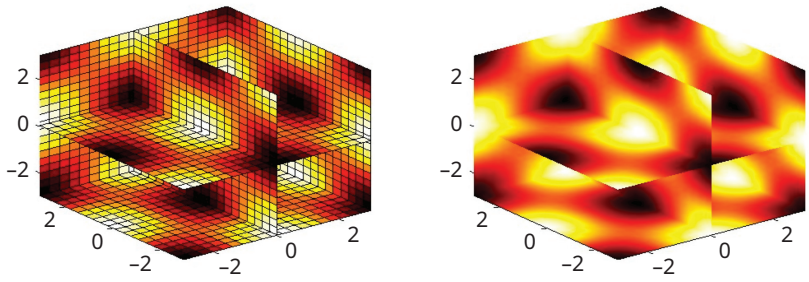


Figure 7.9 Implementation of the **slice** plotting routine for the slice planes at $x = 0, 3$, $y = 3$, and $z = -3, 0$. The only difference between the plots is that the **shading interp** command has been applied to the right figure.

```
subplot(2,2,1)
slice(x,y,z,u,[0 3],3,[0 -3])
colormap(hot)
set(gca,'Xlim',[-3 3],'Ylim',[-3 3],'Zlim',[-3 3], 'Fontsize',[14])
subplot(2,2,2)
slice(x,y,z,u,[0 3],3,[0 -3])
colormap(hot)
shading interp
set(gca,'Xlim',[-3 3],'Ylim',[-3 3],'Zlim',[-3 3], 'Fontsize',[14])
```

In this visualization, several slice planes are used. Specifically, the slice planes are at $x = 0, 3$, $y = 3$, and $z = -3, 0$. Figure 7.9 left, shows the plot without the shading interpolation. The grid lines can often be a nice guide for the eye for generating nice plots. On the right, the shading interpolation is used. Depending on the data, one may want to use either one.

7.3 Movies and Animations

Making movies in Python is a fairly easy proposition. It also affords an excellent opportunity to highlight the hard work accomplished via simulations or experiments. Essentially, the movie-making is based upon using the figures that are generated in Python and using them as frames in a movie.

Two movie files will be exhibited here, one which is a simple implementation that generates a movie file to be read by Python, and a second file which generates a more standard output, namely an AVI file. The example to be considered is the spiral wave of Section 7.1 but with a time-changing phase. Thus, the spiral wave rotates in time. Each frame of the movie is a plot of the spiral wave at a different time-frame.

Two key commands are used in the basic implementation of the movie-making technique: **getframe** and **movie**. The **getframe** command takes the currently plotted figure and takes this to be a frame of the movie. The collection of frames make up the movie. The **movie** command can then run this collection of frames as a movie via Python.

The following code develops a movie by creating frames and saving them into a movie matrix **M**.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from PIL import Image
from IPython.display import Image


L = 10; x = np.linspace(-L, L, 50); y = x
X, Y = np.meshgrid(x, y)

nt=30; frames = []
for j in range(nt):
    u = np.tanh(np.sqrt(X**2 + Y**2))* np.cos((np.angle(X + 1j * Y)
      - np.sqrt(X**2 + Y**2)) + j)

    fig = plt.figure(figsize=(15, 10))
    plt.imshow(u, cmap='coolwarm')
    plt.title(f'Frame {j + 1}/{nt}')
    plt.savefig(f'movie/frame_{j}.png', dpi=80)
        # Adjust dpi as needed
    plt.close()
    frame = Image.open(f'movie/frame_{j}.png')
    frames.append(frame)

frames[0].save('animation.gif', save_all=True,
               append_images=frames[1:], loop=0, duration=100)

Image(filename='animation.gif')  # play movie
```

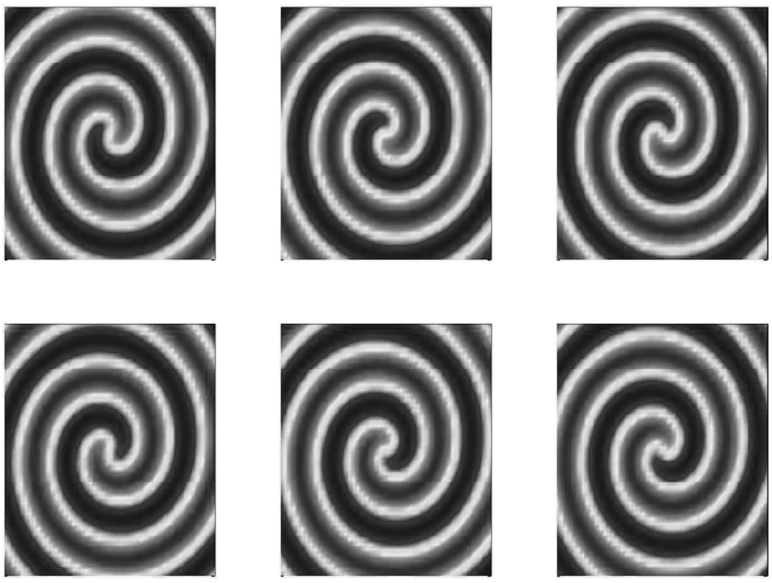


Figure 7.10 The first six frames of the movie file generated from Python. In this movie, the spiral wave rotates in time. Note that for the plot, the tick marks have been removed.

At the end of the code, the **movie(M, 3)** command plays the movie file **M** a total number of three times. The file **M** can be saved for later viewing in Python. Note that the only command in the plot loop that has not been covered previously is the **getframe** command that saves each frame into the jth frame of **M**.

7.3.1 GIF animation files

The movie files saved as in the previous example are good for viewing in Python only. A more general way to save files is in the **.avi** file format. The second example code is a more general, and essentially preferable, way of saving the movie in a format that can be read by many media players outside of Python. The following Python file produces the equivalent of the first code but now in the AVI format.

Note that if the movie is not closed at the end, an error will occur in trying to read the AVI file. In this case, the **mov** file is defined at the onset as an AVI file. The command **'fps'** controls the frames-per-second for the movie. The **addframe** command takes the frames procured from the **getframe** command and adds them to the AVI file MOV. The output of this code is a file called **test.avi**.

The first six frames of the movie generated from either code is illustrated in Fig. 7.10. The tick marks have been removed for convenience. It is observed in this movie that the spiral wave rotates in time. This is very nicely observed from the generated **test.avi** movie file.

PART 2

Differential and Partial Differential Equations

Initial and Boundary Value Problems of Differential Equations

8

8.1 Initial Value Problems: Euler, Runge–Kutta, and Adams Methods 172
8.1.1 Fourth-order Runge–Kutta 175
8.1.2 Adams method: Multi-stepping techniques 175
8.1.3 Higher-order differential equations 178
8.2 Error Analysis for Time-Stepping Routines 179
8.2.1 Round-off and step-size 180
8.2.2 Stability 181
8.3 Advanced Time-Stepping Algorithms 184
8.3.1 Adaptive time-stepping algorithm 184
8.3.2 Exponential time-steppers 185
8.4 Boundary Value Problems: The Shooting Method 187
8.4.1 The shooting method 188
8.4.2 Computational algorithm 189
8.4.3 Shooting example 190
8.4.4 Eigenvalues and eigenfunctions: The infinite domain 192
8.5 Implementation of Shooting and Convergence Studies 195
8.5.1 Convergence 195
8.5.2 Flow control 196
8.6 Boundary Value Problems: Direct Solve and Relaxation 199
8.6.1 Nonlinear systems 202
8.7 Implementing Python for Boundary Value Problems 202
8.7.1 Linear boundary value problem 203
8.7.2 Nonlinear eigenvalue problem 205
8.8 Linear Operators and Computing Spectra 207
8.8.1 Sturm–Liouville theory 209
8.8.2 Derivative operators 209
8.8.3 Example: The harmonic oscillator 213
8.9 Neural Networks for Time-Stepping 214

Our ultimate goal is to solve very general nonlinear partial differential equations of elliptic, hyperbolic, parabolic, or mixed type. However, a variety of basic techniques are required from the solutions of ordinary differential equations. By understanding the basic ideas for computationally solving initial and boundary value problems for differential equations, we

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.
© J. Nathan Kutz (2026). DOI: 10.1093/oso/9780198929093.003.0009

can solve more complicated partial differential equations. The development of numerical solution techniques for initial and boundary value problems originates from the simple concept of the Taylor expansion. Thus, the building blocks for scientific computing are rooted in concepts from freshman calculus. Implementation, however, often requires ingenuity, insight, and clever application of the basic principles. In some sense, our numerical solution techniques reverse our understanding of calculus. Whereas calculus teaches us to take a limit in order to define a derivative or integral, in numerical computations, we take the derivative or integral of the governing equation and go backwards to define it as the difference.

8.1 Initial Value Problems: Euler, Runge–Kutta, and Adams Methods

The solutions of general partial differential equations rely heavily on the techniques developed for ordinary differential equations. Thus, we begin by considering systems of differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) \tag{8.1.1}$$

where $\mathbf{y}$ represents the solution vector of interest and the general function $f(t, \mathbf{y})$ models the specific system of interest. Indeed, the function $f(t, \mathbf{y})$ is the primary quantity required for calculating the dynamical evolution of a given system. As such, it will be the critical part of building Python codes for solving differential equations. In addition to the evolution dynamics, the initial conditions are given by

$$\mathbf{y}(0) = \mathbf{y}_0 \tag{8.1.2}$$

with $t \in [0, T]$. Although very simple in appearance, this equation cannot be solved analytically in general. Of course, there are certain cases for which the problem can be solved analytically, but it will generally be important to rely on numerical solutions for insight. For an overview of analytic techniques, see Boyce and DiPrima [29]. Note that the function $f(\mathbf{y}, t)$ is what is ultimately required by Python to solve a given differential equation system.

The simplest algorithm for solving this system of differential equations is known as the *Euler method*. The Euler method is derived by making use of the definition of the derivative:

$$\frac{d\mathbf{y}}{dt} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{y}}{\Delta t}. \tag{8.1.3}$$

Thus, over a time-span $\Delta t = t_{n+1} - t_n$, we can approximate the original differential equation by

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) \quad \Rightarrow \quad \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} \approx f(t_n, \mathbf{y}_n). \tag{8.1.4}$$

The approximation can easily be rearranged to give

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n, \mathbf{y}_n). \tag{8.1.5}$$

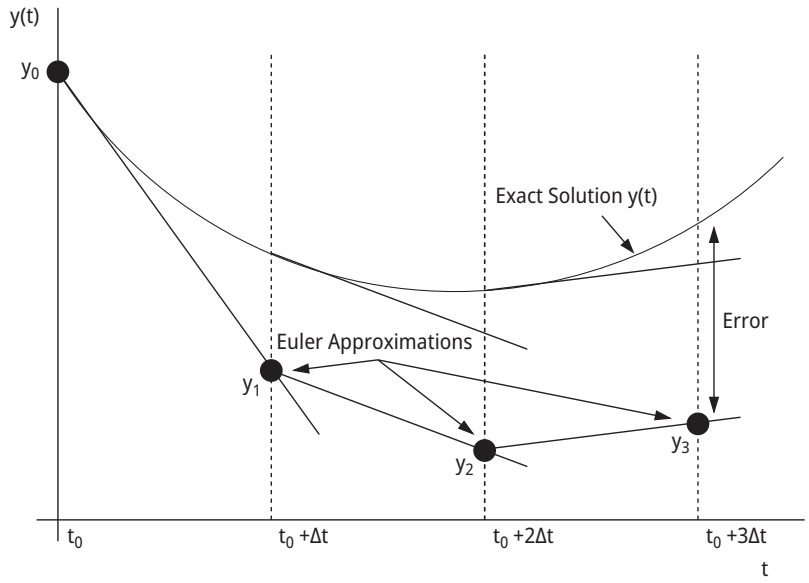


Figure 8.1 Graphical description of the iteration process used in the Euler method. Note that each subsequent approximation is generated from the slope of the previous point. This graphical illustration suggests that smaller steps Δt should be more accurate.

Thus, the Euler method gives an iterative scheme by which the future values of the solution can be determined. Generally, the algorithm structure is of the form

$$\mathbf{y}(t_{n+1}) = F(\mathbf{y}(t_n)) \tag{8.1.6}$$

where $F(\mathbf{y}(t_n)) = \mathbf{y}(t_n) + \Delta t \cdot f(t_n, \mathbf{y}(t_n))$. The graphical representation of this iterative process is illustrated in Fig. 8.1, where the slope (derivative) of the function is responsible for generating each subsequent approximation to the solution $\mathbf{y}(t)$. Note that the Euler method is exact as the step-size decreases to zero: $\Delta t \to 0$.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \phi \tag{8.1.7}$$

where the function ϕ is chosen to reduce the error over a single time-step Δt and $\mathbf{y}_n = \mathbf{y}(t_n)$. The function ϕ is no longer constrained, as in the Euler scheme, to make use of the derivative at the left end point of the computational step. Rather, the derivative at the midpoint of the time-step and at the right end of the time-step may also be used to possibly improve accuracy. In particular, by generalizing to include the slope at the left and right ends of the time-step Δt, we can generate an iteration scheme of the following form:

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t \left[A f(t, \mathbf{y}(t)) + B f\left(t + P \cdot \Delta t, \mathbf{y}(t) + Q \Delta t \cdot f(t, \mathbf{y}(t))\right)\right] \tag{8.1.8}$$

where A, B, P, and Q are arbitrary constants. Upon Taylor expanding the last term, we find

$$f(t + P \cdot \Delta t, \mathbf{y}(t) + Q\Delta t \cdot f(t, \mathbf{y}(t))) = f(t, \mathbf{y}(t)) + P\Delta t \cdot f_t(t, \mathbf{y}(t)) + Q\Delta t \cdot f_{\mathbf{y}}(t, \mathbf{y}(t)) \cdot f(t, \mathbf{y}(t)) + O(\Delta t^2) \quad (8.1.9)$$

where f_t and $f_{\mathbf{y}}$ denote differentiation with respect to t and $\mathbf{y}$, respectively, use has been made of (8.1.1), and $O(\Delta t^2)$ denotes all terms that are of size Δt^2 and smaller. Plugging in this last result into the original iteration scheme (8.1.8) results in the following:

$$\begin{aligned} \mathbf{y}(t + \Delta t) = \mathbf{y}(t) &+ \Delta t(A + B) f(t, \mathbf{y}(t)) \\ &+ PB\Delta t^2 \cdot f_t(t, \mathbf{y}(t)) \\ &+ BQ\Delta t^2 \cdot f_{\mathbf{y}}(t, \mathbf{y}(t)) \cdot f(t, \mathbf{y}(t)) + O\left(\Delta t^3\right) \end{aligned} \quad (8.1.10)$$

which is valid up to $O(\Delta t^2)$.

To proceed further, we simply note that the Taylor expansion for $\mathbf{y}(t + \Delta t)$ gives:

$$\begin{aligned} \mathbf{y}(t + \Delta t) = \mathbf{y}(t) &+ \Delta t \cdot f(t, \mathbf{y}(t)) + \frac{1}{2}\Delta t^2 \cdot f_t(t, \mathbf{y}(t)) \\ &+ \frac{1}{2}\Delta t^2 \cdot f_{\mathbf{y}}\left(t, \mathbf{y}(t)\right) f\left(t, \mathbf{y}(t)\right) + O\left(\Delta t^3\right). \end{aligned} \quad (8.1.11)$$

Comparing this Taylor expansion with (8.1.10) gives the following relations:

$$A + B = 1 \quad (8.1.12a)$$

$$PB = \frac{1}{2} \quad (8.1.12b)$$

$$BQ = \frac{1}{2} \quad (8.1.12c)$$

which yields three equations for the four unknowns A, B, P, and Q. Thus, one degree of freedom is granted, and a wide variety of schemes can be implemented. Two of the more commonly used schemes are known as *Heun's method* and *modified Euler–Cauchy* (second-order Runge–Kutta). These schemes assume $A = 1/2$ and $A = 0$, respectively, and are given by:

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \frac{\Delta t}{2}\left[f(t, \mathbf{y}(t)) + f(t + \Delta t, \mathbf{y}(t) + \Delta t \cdot f(t, \mathbf{y}(t)))\right] \quad (8.1.13a)$$

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t \cdot f\left(t + \frac{\Delta t}{2}, \mathbf{y}(t) + \frac{\Delta t}{2} \cdot f(t, \mathbf{y}(t))\right). \quad (8.1.13b)$$

Generally speaking, these methods for iterating forward in time given a single initial point are known as *Runge–Kutta methods*. By generalizing the assumption (8.1.8), we can construct stepping schemes which have arbitrary accuracy. Of course, the level of algebraic difficulty in deriving these higher accuracy schemes also increases significantly from Heun's method and modified Euler–Cauchy.

8.1.1 Fourth-order Runge–Kutta

Perhaps the most popular general stepping scheme used in practice is known as the *fourth-order Runge–Kutta method*. The term "fourth-order" refers to the fact that the Taylor series local truncation error is pushed to $O(\Delta t^5)$. The total cumulative (global) error is then $O(\Delta t^4)$ and is responsible for the scheme name of "fourth-order". The scheme is as follows:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{6}\left[f_1 + 2f_2 + 2f_3 + f_4\right] \tag{8.1.14}$$

where

$$f_1 = f(t_n, \mathbf{y}_n) \tag{8.1.15a}$$

$$f_2 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_1\right) \tag{8.1.15b}$$

$$f_3 = f\left(t_n + \frac{\Delta t}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f_2\right) \tag{8.1.15c}$$

$$f_4 = f\left(t_n + \Delta t, \mathbf{y}_n + \Delta t \cdot f_3\right). \tag{8.1.15d}$$

This scheme gives a local truncation error which is $O(\Delta t^5)$. The cumulative (global) error in this case is fourth order so that for $t \sim O(1)$, the error is $O(\Delta t^4)$. The key to this method, as well as any of the other Runge–Kutta schemes, is the use of intermediate time-steps to improve accuracy. For the fourth-order scheme presented here, a graphical representation of this derivative sampling at intermediate time-steps is shown in Fig. 8.2.

8.1.2 Adams method: Multi-stepping techniques

The development of the Runge–Kutta schemes relies on the definition of the derivative and Taylor expansions. Another approach to solving (8.1.1) is to start with the fundamental

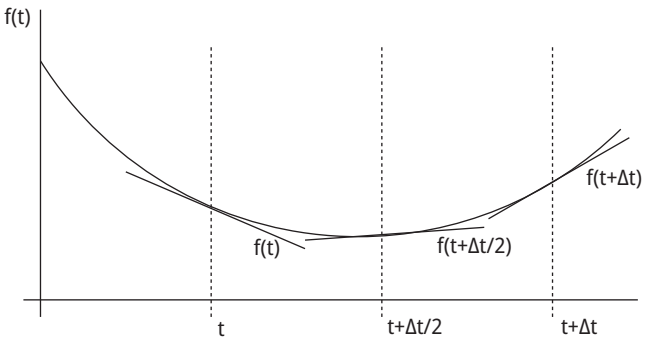


Figure 8.2 Graphical description of the initial, intermediate, and final slopes used in the fourth-order Runge–Kutta iteration scheme over a time Δt.

theorem of calculus [30]. Thus, the differential equation can be integrated over a time-step Δt to give

$$\frac{d\mathbf{y}}{dt} = f(t,\mathbf{y}) \quad \Rightarrow \quad \mathbf{y}(t+\Delta t) - \mathbf{y}(t) = \int_t^{t+\Delta t} f(t,\mathbf{y})dt. \tag{8.1.16}$$

And once again using our iteration notation, we find

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(t,\mathbf{y})dt. \tag{8.1.17}$$

This integral iteration relation is simply a restatement of (8.1.7) with $\Delta t \cdot \phi = \int_{t_n}^{t_{n+1}} f(t,\mathbf{y})dt$. However, at this point, no approximations have been made and (8.1.17) is exact. The numerical solution will be found by approximating $f(t,\mathbf{y}) \approx p(t,\mathbf{y})$ where $p(t,\mathbf{y})$ is a polynomial. Thus, the iteration scheme in this instance will be given by

$$\mathbf{y}_{n+1} \approx \mathbf{y}_n + \int_{t_n}^{t_{n+1}} p(t,\mathbf{y})dt. \tag{8.1.18}$$

It only remains to determine the form of the polynomial to be used in the approximation.

The *Adams–Bashforth* suite of computational methods uses the current point and a determined number of past points to evaluate the future solution. As with the Runge–Kutta schemes, the order of accuracy is determined by the choice of ϕ. In the Adams–Bashforth case, this relates directly to the choice of the polynomial approximation $p(t,\mathbf{y})$. A first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_n,\mathbf{y}_n), \tag{8.1.19}$$

where the present point and no past points are used to determine the value of the polynomial. Inserting this first-order approximation into (8.1.18) results in the previously found Euler scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_n,\mathbf{y}_n). \tag{8.1.20}$$

Alternatively, we could assume that the polynomial used both the current point and the previous point so that a second-order scheme resulted. The linear polynomial which passes through these two points is given by

$$p_2(t) = f_{n-1} + \frac{f_n - f_{n-1}}{\Delta t}(t - t_{n-1}). \tag{8.1.21}$$

When inserted into (8.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_n - f_{n-1}}{\Delta t}(t - t_{n-1}) \right) dt. \tag{8.1.22}$$

Upon integration and evaluation at the upper and lower limits, we find the following second-order Adams–Bashforth scheme:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[3f\left(t_n, \mathbf{y}_n\right) - f\left(t_{n-1}, \mathbf{y}_{n-1}\right)\right]. \tag{8.1.23}$$

In contrast to the Runge–Kutta method, this is a *two-step algorithm* which requires two initial conditions. This technique can be easily generalized to include more past points and thus higher accuracy. However, as accuracy is increased, so are the number of initial conditions required to step forward one time-step Δt. Aside from the first-order accurate scheme, any implementation of the Adams–Bashforth scheme will require a *bootstrap* to generate a second "initial condition" for the solution iteration process.

The Adams–Bashforth scheme uses current and past points to approximate the polynomial $p(t, \mathbf{y})$ in (8.1.18). If instead a future point, the present, and the past is used, then the scheme is known as an *Adams–Moulton method*. As before, a first-order scheme can easily be constructed by allowing

$$p_1(t) = \text{constant} = f(t_{n+1}, \mathbf{y}_{n+1}), \tag{8.1.24}$$

where the future point and no past and present points are used to determine the value of the polynomial. Inserting this first-order approximation into (8.1.18) results in the *backward Euler scheme*

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f(t_{n+1}, \mathbf{y}_{n+1}). \tag{8.1.25}$$

Alternatively, we could assume that the polynomial used both the future point and the current point so that a second-order scheme resulted. The linear polynomial which passes through these two points is given by

$$p_2(t) = f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n). \tag{8.1.26}$$

Inserted into (8.1.18), this linear polynomial yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{\Delta t}(t - t_n)\right) dt. \tag{8.1.27}$$

Upon integration and evaluation at the upper and lower limits, we find the following second-order Adams–Moulton scheme:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[f(t_{n+1}, \mathbf{y}_{n+1}) + f(t_n, \mathbf{y}_n)\right]. \tag{8.1.28}$$

Once again this is a two-step algorithm. However, it is categorically different from the Adams–Bashforth methods since it results in an *implicit scheme*, i.e. the unknown value $\mathbf{y}_{n+1}$ is specified through a nonlinear equation (8.1.28). The solution of this nonlinear system can be very difficult, thus making *explicit schemes* such as Runge–Kutta and Adams–Bashforth,

which are simple iterations, more easily handled. However, implicit schemes can have advantages when considering stability issues related to time-stepping.

One way to circumvent the difficulties of the implicit stepping method while still making use of its power is to use a *predictor–corrector method*. This scheme draws on the power of both the Adams–Bashforth and Adams–Moulton schemes. In particular, the second-order implicit scheme given by (8.1.28) requires the value of $f(t_{n+1}, \mathbf{y}_{n+1})$ in the right-hand side. If we can predict (approximate) this value, then we can use this predicted value to solve (8.1.28) explicitly. Thus, we begin with a predictor step to estimate $\mathbf{y}_{n+1}$ so that $f(t_{n+1}, \mathbf{y}_{n+1})$ can be evaluated. We then insert this value into the right-hand side of (8.1.28) and explicitly find the corrected value of $\mathbf{y}_{n+1}$. The second-order predictor–corrector steps are then as follows:

$$\text{Predictor (Adams–Bashforth): } \mathbf{y}_{n+1}^{P} = \mathbf{y}_n + \frac{\Delta t}{2}\left[3f_n - f_{n-1}\right] \tag{8.1.29a}$$

$$\text{Corrector (Adams–Moulton): } \mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{2}\left[f\left(t_{n+1}, \mathbf{y}_{n+1}^{P}\right) + f(t_n, \mathbf{y}_n)\right]. \tag{8.1.29b}$$

Thus, the scheme utilizes both explicit and implicit time-stepping schemes without having to solve a system of nonlinear equations.

8.1.3 Higher-order differential equations

Thus far, we have considered systems of first-order equations. Higher-order differential equations can be put into this form, and the methods outlined here can be applied. For example, consider the third-order, nonhomogeneous, differential equation

$$\frac{d^3u}{dt^3} + u^2\frac{du}{dt} + \cos t \cdot u = g(t). \tag{8.1.30}$$

By defining

$$y_1 = u \tag{8.1.31a}$$

$$y_2 = \frac{du}{dt} \tag{8.1.31b}$$

$$y_3 = \frac{d^2u}{dt^2}, \tag{8.1.31c}$$

we find that $dy_3/dt = d^3u/dt^3$. Using the original equation along with the definitions of y_i, we find that

$$\frac{dy_1}{dt} = y_2 \tag{8.1.32a}$$

$$\frac{dy_2}{dt} = y_3 \tag{8.1.32b}$$

$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2\frac{du}{dt} - \cos t \cdot u + g(t) = -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \tag{8.1.32c}$$

which results in the original differential equation (8.1.1) considered previously

$$\frac{d\mathbf{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t \cdot y_1 + g(t) \end{pmatrix} = f(t, \mathbf{y}). \tag{8.1.33}$$

At this point, all the time-stepping techniques developed thus far can be applied to the problem. It is imperative to write any differential equation as a first-order system before solving it numerically with the time-stepping schemes developed here.

8.2 Error Analysis for Time-Stepping Routines

Accuracy and *stability* are fundamental to numerical analysis and are the key factors in evaluating any numerical integration technique. Therefore, it is essential to evaluate the accuracy and stability of the time-stepping schemes developed. Rarely does it occur that both accuracy and stability work in concert. In fact, they are often offsetting and work directly against each other. Thus, a highly accurate scheme may compromise stability, whereas a low-accuracy scheme may have excellent stability properties.

We begin by exploring accuracy. In the context of time-stepping schemes, the natural place to begin is with Taylor expansions. Thus, we consider the expansion

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \Delta t \cdot \frac{d\mathbf{y}(t)}{dt} + \frac{\Delta t^2}{2} \cdot \frac{d^2\mathbf{y}(c)}{dt^2} \tag{8.2.1}$$

where $c \in [t, t + \Delta t]$. Since we are considering $d\mathbf{y}/dt = f(t, \mathbf{y})$, the above formula reduces to the Euler iteration scheme

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot f\left(t_n, \mathbf{y}_n\right) + O\left(\Delta t^2\right). \tag{8.2.2}$$

It is clear from this that the truncation error is $O(\Delta t^2)$. Specifically, the truncation error is given by $\Delta t^2/2 \cdot d^2\mathbf{y}(c)/dt^2$.

Of importance is how this truncation error contributes to the overall error in the numerical solution. Two types of error are important to identify: *local* and *global error*. Each is significant in its own right. However, in practice, we are only concerned with the global (cumulative) error. The *global discretization error* is given by

$$E_k = \mathbf{y}(t_k) - \mathbf{y}_k \tag{8.2.3}$$

where $\mathbf{y}(t_k)$ is the exact solution and $\mathbf{y}_k$ is the numerical solution. The *local discretization error* is given by

$$\epsilon_{k+1} = \mathbf{y}(t_{k+1}) - (\mathbf{y}(t_k) + \Delta t \cdot \phi) \tag{8.2.4}$$

where $\mathbf{y}(t_{k+1})$ is the exact solution and $\mathbf{y}(t_k) + \Delta t \cdot \phi$ is a one-step approximation over the time interval $t \in [t_n, t_{n+1}]$.

Table 8.1 Local and global discretization errors associated with various time-stepping schemes.

scheme	local error ϵ_k	global error E_k
Euler	$O(\Delta t^2)$	$O(\Delta t)$
2nd-order Runge–Kutta	$O(\Delta t^3)$	$O(\Delta t^2)$
4th-order Runge–Kutta	$O(\Delta t^5)$	$O(\Delta t^4)$
2nd-order Adams–Bashforth	$O(\Delta t^3)$	$O(\Delta t^2)$

For the Euler method, we can calculate both the local and global error. Given a time-step Δt and a specified time interval $t \in [a, b]$, we have after K steps that $\Delta t \cdot K = b - a$. Thus, we find

$$\text{local:} \quad \epsilon_k = \frac{\Delta t^2}{2} \frac{d^2\mathbf{y}(c_k)}{dt^2} \sim O(\Delta t^2) \tag{8.2.5a}$$

$$\begin{aligned} \text{global:} \quad E_k &= \sum_{j=1}^{K} \frac{\Delta t^2}{2} \frac{d^2\mathbf{y}(c_j)}{dt^2} \approx \frac{\Delta t^2}{2} \frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot K \\ &= \frac{\Delta t^2}{2} \frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \cdot \frac{b-a}{\Delta t} = \frac{b-a}{2} \Delta t \cdot \frac{d^2\mathbf{y}(\mathbf{c})}{dt^2} \sim O(\Delta t) \end{aligned} \tag{8.2.5b}$$

which gives a local error for the Euler scheme which is $O(\Delta t^2)$ and a global error which is $O(\Delta t)$. Thus, the cumulative error is large for the Euler scheme, i.e. it is not very accurate.

A similar procedure can be carried out for all the schemes discussed thus far, including the multi-step Adams schemes. Table 8.1 illustrates various schemes and their associated local and global errors. The error analysis suggests that the error will always decrease in some power of Δt. Thus, it is tempting to conclude that higher accuracy is easily achieved by taking smaller time-steps Δt. This would be true if not for round-off error in the computer.

8.2.1 Round-off and step-size

An unavoidable consequence of working with numerical computations is round-off error. When working with most computations, *double precision* numbers are used. This allows for 16-digit accuracy in the representation of a given number. This round-off has a significant impact upon numerical computations and the issue of time-stepping.

As an example of the impact of round-off, we consider the Euler approximation to the derivative

$$\frac{d\mathbf{y}}{dt} \approx \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} + \epsilon(\mathbf{y}_n, \Delta t) \tag{8.2.6}$$

where $\epsilon(\mathbf{y}_n, \Delta t)$ measures the truncation error. Upon evaluating this expression in the computer, round-off error occurs so that

$$\mathbf{y}_{n+1} = \mathbf{Y}_{n+1} + \mathbf{e}_{n+1}. \tag{8.2.7}$$

Thus, the combined error between the round-off and truncation gives the following expression for the derivative:

$$\frac{d\mathbf{y}}{dt} = \frac{\mathbf{Y}_{n+1} - \mathbf{Y}_n}{\Delta t} + E_n(\mathbf{y}_n, \Delta t) \tag{8.2.8}$$

where the total error, E_n, is the combination of round-off and truncation such that

$$E_n = E_{\text{round}} + E_{\text{trunc}} = \frac{\mathbf{e}_{n+1} - \mathbf{e}_n}{\Delta t} - \frac{\Delta t}{2}\frac{d^2\mathbf{y}(c)}{dt^2}. \tag{8.2.9}$$

We now determine the maximum size of the error. In particular, we can bound the maximum value of round-off and the second derivative to be

$$|\mathbf{e}_{n+1}| \leq e_r \tag{8.2.10a}$$

$$|-\mathbf{e}_n| \leq e_r \tag{8.2.10b}$$

$$M = \max_{c\in[t_n,t_{n+1}]}\left\{\left|\frac{d^2\mathbf{y}(c)}{dt^2}\right|\right\}. \tag{8.2.10c}$$

This then gives the maximum error to be

$$|E_n| \leq \frac{e_r + e_r}{\Delta t} + \frac{\Delta t}{2}M = \frac{2e_r}{\Delta t} + \frac{\Delta t M}{2}. \tag{8.2.11}$$

To minimize the error, we require that $\partial|E_n|/\partial(\Delta t) = 0$. Calculating this derivative gives

$$\frac{\partial|E_n|}{\partial(\Delta t)} = -\frac{2e_r}{\Delta t^2} + \frac{M}{2} = 0, \tag{8.2.12}$$

so that

$$\Delta t = \left(\frac{4e_r}{M}\right)^{1/2}. \tag{8.2.13}$$

This gives the step-size resulting in a minimum error. Thus, the smallest step-size is not necessarily the most accurate. Rather, a balance between round-off error and truncation error is achieved to obtain the optimal step-size.

8.2.2 Stability

The accuracy of any scheme is certainly important. However, it is meaningless if the scheme is not stable numerically. The essence of a stable scheme: the numerical solutions do not blow up to infinity. As an example, consider the simple differential equation

$$\frac{dy}{dt} = \lambda y \tag{8.2.14}$$

with

$$y(0) = y_0. \tag{8.2.15}$$

The analytic solution is easily calculated to be $y(t) = y_0 \exp(\lambda t)$. However, if we solve this problem numerically with a forward Euler method, we find

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_n = (1 + \lambda \Delta t) y_n. \tag{8.2.16}$$

After N steps, we find this iteration scheme yields

$$y_N = (1 + \lambda \Delta t)^N y_0. \tag{8.2.17}$$

Given that we have a certain amount of round-off error, the numerical solution would then be given by

$$y_N = (1 + \lambda \Delta t)^N (y_0 + e). \tag{8.2.18}$$

The error then associated with this scheme is given by

$$E = (1 + \lambda \Delta t)^N e. \tag{8.2.19}$$

At this point, the following observations can be made. For $\lambda > 0$, the solution $y_N \to \infty$ in (8.2.18) as $N \to \infty$. So although the error also grows, it may not be significant in comparison to the size of the numerical solution.

In contrast, equation (8.2.18) for $\lambda < 0$ is markedly different. For this case, $y_N \to 0$ in (8.2.18) as $N \to \infty$. The error, however, can dominate in this case. In particular, we have the following two cases for the error given by (8.2.19):

$$\text{I:} \quad |1 + \lambda \Delta t| < 1 \quad \text{then} \quad E \to 0 \tag{8.2.20a}$$

$$\text{II:} \quad |1 + \lambda \Delta t| > 1 \quad \text{then} \quad E \to \infty. \tag{8.2.20b}$$

In case I, the scheme would be considered stable. However, case II holds and is unstable provided $\Delta t > -2/\lambda$.

A general theory of stability can be developed for any one-step time-stepping scheme. Consider the one-step recursion relation for an $M \times M$ system

$$\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n. \tag{8.2.21}$$

After N steps, the algorithm yields the solution

$$\mathbf{y}_N = \mathbf{A}^N \mathbf{y}_0, \tag{8.2.22}$$

where $\mathbf{y}_0$ is the initial vector. A well-known result from linear algebra is that

$$\mathbf{A}^N = \mathbf{S}\mathbf{\Lambda}^N\mathbf{S}^{-1} \tag{8.2.23}$$

where $\mathbf{S}$ is the matrix whose columns are the eigenvectors of $\mathbf{A}$, and

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M \end{pmatrix} \rightarrow \mathbf{\Lambda}^N = \begin{pmatrix} \lambda_1^N & 0 & \cdots & 0 \\ 0 & \lambda_2^N & 0 & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda_M^N \end{pmatrix} \tag{8.2.24}$$

is a diagonal matrix whose entries are the eigenvalues of $\mathbf{A}$. Thus, upon calculating $\mathbf{\Lambda}^N$, we are only concerned with the eigenvalues. In particular, instability occurs if $|\lambda_i| > 1$ for $i = 1, 2, \ldots, M$. This method can be easily generalized to two-step schemes (Adams methods) by considering $\mathbf{y}_{n+1} = \mathbf{A}\mathbf{y}_n + \mathbf{B}\mathbf{y}_{n-1}$.

Lending further significance to this stability analysis is its connection with practical implementation. We contrast the difference in stability between the forward and backward Euler schemes. The forward Euler scheme has already been considered in (8.2.16)–(8.2.19). The backward Euler displays significant differences in stability. If we again consider (8.2.14) with (8.2.15), the backward Euler method gives the iteration scheme

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_{n+1}, \tag{8.2.25}$$

which after N steps leads to

$$y_N = \left(\frac{1}{1 - \lambda\Delta t}\right)^N y_0. \tag{8.2.26}$$

The round-off error associated with this scheme is given by

$$E = \left(\frac{1}{1 - \lambda\Delta t}\right)^N e. \tag{8.2.27}$$

By letting $z = \lambda\Delta t$ be a complex number, we find the following criteria to yield unstable behaviour based upon (8.2.19) and (8.2.27):

$$\text{Forward Euler:} \quad |1 + z| > 1 \tag{8.2.28a}$$

$$\text{Backward Euler:} \quad \left|\frac{1}{1 - z}\right| > 1. \tag{8.2.28b}$$

Figure 8.3 shows the regions of stable and unstable behaviour as a function of z. It is observed that the forward Euler scheme has a very small range of stability, whereas the backward Euler scheme has a large range of stability. This large stability region is part of what makes implicit methods so attractive. Thus, stability regions can be calculated. However, control of the accuracy is also essential.

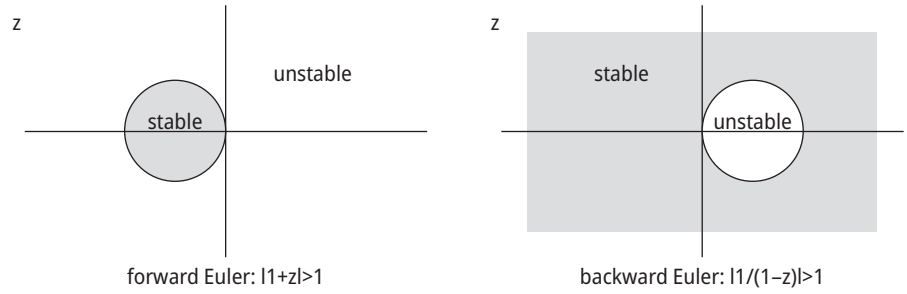


Figure 8.3 Regions for stable stepping (shaded) for the forward Euler and backward Euler schemes. The criteria for instability is also given for each stepping method.

8.3 **Advanced Time-Stepping Algorithms**

Before closing our analysis on time-stepping algorithms, a few alternative methods are considered in the interest of achieving better performance. Specifically, the commonly used adaptive time-stepping algorithm will be outlined. In addition, the exponential time-stepper will be developed for numerically stiff problems.

8.3.1 Adaptive time-stepping algorithm

Adaptive time-stepping algorithms are tremendously important in practice. The premise of the adaptive stepping technique is to take as large a time-step as possible while retaining a prescribed accuracy. All of the time-stepping algorithms used in Python have a built-in adaptive stepper. Indeed, the fundamental premise of the standard differential equation solver in Python is to guarantee a solution with a prescribed absolute and relative tolerance. The time-step Δt is chosen so that the tolerance constraints are met.

The following algorithm outlines the method employed in the adaptive stepping method routine.

(1) Start with a default time-step Δt_0.

(2) Use one of the iteration algorithms (such as fourth-order Runge–Kutta) to take a time-step Δt into the future. Represent the solution by $f_1(t + \Delta t)$. The initial $\Delta t = \Delta t_0$.

(3) Now cut the time-step in half ($\Delta t/2$) and use the iteration algorithm to advance Δt into the future. This would require two iterative steps. Represent the solution by $f_2(t + \Delta t)$.

(4) Compare the solutions using Δt and $\Delta t/2$. For instance, one may measure the L^2 norm between the two solutions: $E = \|f_1 - f_2\|$.

(5) If the difference is above a prescribed tolerance, i.e. $E >$ tolerance, then cut the time-step in half again to $\Delta t/4$ and compare again. Continue cutting the time-step in half until the tolerance is achieved.

(6) If the comparison is already below the prescribed tolerance, then the time-step can be doubled in size to $2\Delta t$. The comparison can be made again until the tolerance condition is violated.

This algorithm is a shell of what the algorithm might look like. Certainly a more sophisticated version can be constructed, but this illustrates the key concept of either making the time-step bigger or smaller as needed.

The advantages of such a scheme are enormous. It allows the iterative method for advancing the differential equation solution into the future to be maximally efficient. When the solution is changing slowly, the time-steps will be quite large, whereas when the solution changes rapidly, the time-step will automatically adjust and shorten in order to preserve the accuracy.

In Python, the time-step can be adjusted by modifying the tolerance settings associated with the time-stepping algorithm. The following code adjusts the time-step so that a 10^{-4} accuracy is achieved both for relative tolerance and absolute tolerance.

```
TOL=1e-4; OPTIONS = odeset('RelTol',TOL,'AbsTol',TOL);
[t,y] = ode45('F',tspan,y0,OPTIONS);
```

The speed of the code is largely determined by the accuracy setting as determined from the **odeset** command. The default is a 10^{-6} tolerance for both relative and absolute error.

8.3.2 Exponential time-steppers

A nice example of a time-stepping technique that removes numerical stiffness generated from either large linear terms (or linear, high-order derivative terms) is the exponential time-stepping technique [31; 32]. This is the only stiff time-stepper that will be considered in detail in this book. As with any other stiff-stepping technique, advantage is taken of certain properties of the differential equation considered in order to make the algorithm faster, i.e. in order to maximize the step-size while keeping a fixed accuracy.

The prototype equation to be considered is the differential equation of the form [31]

$$\frac{d\mathbf{y}}{dt} = c\mathbf{y} + F(\mathbf{y}, t) \tag{8.3.1}$$

where $|c| \gg 1$. When c is large in magnitude, it dominates the selection of the time-step Δt. In fact, it forces the time-step Δt to be quite small in order to accurately resolve the future solution $\mathbf{y}(t + \Delta t)$. It should be noted that the large c term often arises from high-order and linear derivatives in problems involving partial differential equations. This will be considered in future sections.

One method of dealing with numerical stiffness induced by c is to attempt a solution via the integrating factor method. Multiplying (8.3.1) by the factor $\exp(-ct)$ gives the following set of algebraic reductions:

$$\begin{aligned}\frac{d\mathbf{y}}{dt}\exp(-ct) &= c\mathbf{y}\exp(-ct) + F(\mathbf{y},t)\exp(-ct)\\ \frac{d\mathbf{y}}{dt}\exp(-ct) - c\mathbf{y}\exp(-ct) &= F(\mathbf{y},t)\exp(-ct)\\ \frac{d}{dt}\left(\mathbf{y}\exp(-ct)\right) &= F(\mathbf{y},t)\exp(-ct).\end{aligned} \tag{8.3.2}$$

Integrating both sides from time t to time $t+\Delta t$ yields

$$\mathbf{y}(t+\Delta t)\exp\left(-c(t+\Delta t)\right) - \mathbf{y}(t)\exp(-ct) = \int_t^{t+\Delta t} F(\mathbf{y}(\tau),\tau)\exp(-c\tau)d\tau. \tag{8.3.3}$$

Finally, by multiplying both sides by $\exp(c(t+\Delta t))$ and making a change of variables in the integral, the following formula is achieved:

$$\mathbf{y}(t+\Delta t) = \mathbf{y}(t)\exp(c\Delta t) + \exp(c\Delta t)\int_0^{\Delta t} F(\mathbf{y}(t+\tau),t+\tau)\exp(-c\tau)d\tau. \tag{8.3.4}$$

This formula is exact. Moreover, it has some of the basic characteristics of the Adams–Bashforth and Adams–Moulton schemes considered earlier, where the integral approximation determines the accuracy and iteration of the scheme. But unlike the Adams methods, the linear term that is scaled with the parameter c is explicitly accounted for by the integrating factor.

The approximation of the integral yields the exponential steppers of interest. The simplest approximation is to assume that the function F takes on a constant value so that $F(\mathbf{y}(t+\tau),t+\tau) \approx F(\mathbf{y}(t),t)$. Integrating the integral now with respect to τ yields

$$\mathbf{y}_{n+1} = \mathbf{y}_n\exp(c\Delta t) + F_n\left(\exp(c\Delta t) - 1\right)/c \tag{8.3.5}$$

where $\mathbf{y}_n = \mathbf{y}(t_n)$ and $F_n = F(\mathbf{y}_n,t_n)$. Note that in the limit as $c \ll 1$, this formula asymptotically approaches the Euler stepping formula.

As with the Adams–Bashforth method, a higher-order approximation can be used for evaluating the integral. In particular, the following can be used:

$$F = F_n + \tau\left(F_n - F_{n-1}\right)/\Delta t + O\left(\Delta t^2\right). \tag{8.3.6}$$

This approximation to the integrand gives an improved accuracy to the time-stepping method. When inserted into the formula (8.3.4), the following time-stepping algorithm is derived:

$$\begin{aligned}\mathbf{y}_{n+1} = \mathbf{y}_n\exp(c\Delta t) &+ F_n\left[(1+c\Delta t)\exp(c\Delta t) - 1 - 2c\Delta t\right]/\left(c^2\Delta t\right)\\ &+ F_{n-1}\left[1 + c\Delta t - \exp(c\Delta t)\right]/\left(c^2\Delta t\right).\end{aligned} \tag{8.3.7}$$

In the limit as $c \to 0$, this reduces to the second-order Adams–Bashforth scheme. But recall that the purpose of this scheme is to consider problems for which $c \gg 1$.

Cox and Matthews [31] continue with this idea in order to derive the equivalent of the fourth-order Runge–Kutta scheme with the exponential time-stepping explicitly accounted for. The claim is that this derivation is nontrivial and requires careful manipulation and the aid of symbolic computing. Regardless, the following exponential, fourth-order Runge–Kutta scheme is developed:

$$\begin{aligned}\mathbf{y}_{n+1} = e^{c\Delta t}\mathbf{y}_n &+ \left[-4 - c\Delta t + e^{c\Delta t}\left(4 - 3c\Delta t + (c\Delta t)^2\right)\right] F(\mathbf{y}_n, t_n) \,/\, \left(c^3\Delta t^2\right) \\ &+ 2\left[2 + c\Delta t + e^{c\Delta t}(-2 + c\Delta t)\right] F(\mathbf{a}_n, t_n + \Delta t/2) + F(\mathbf{b}_n, t_n + \Delta t/2) \\ &+ \left[-4 - 3c\Delta t - (c\Delta t)^2 + e^{c\Delta t}\left(4 - c\Delta t\right)\right] F(\mathbf{c}_n, t_n + \Delta t) \end{aligned} \tag{8.3.8}$$

where

$$\mathbf{a}_n = \mathbf{y}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) F(\mathbf{y}_n, t_n)/c \tag{8.3.9a}$$

$$\mathbf{b}_n = \mathbf{y}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) F(\mathbf{a}_n, t_n + \Delta t/2)/c \tag{8.3.9b}$$

$$\mathbf{c}_n = \mathbf{a}_n e^{c\Delta t/2} + \left(e^{c\Delta t/2} - 1\right) \left[2F(\mathbf{b}_n, t_n + \Delta t/2) - F(\mathbf{y}_n, t_n)\right]/c. \tag{8.3.9c}$$

To implement this in practice, Kassam and Trefethen [32] show that special care must be taken in order to evaluate the coefficients $\mathbf{a}_n, \mathbf{b}_n$, and $\mathbf{c}_n$. This evaluation is intimately related to the well-known numerical difficulty in evaluating the function $(e^z - 1)/z$. However, using the method of Kassam and Trefethen [32], the exponential time-stepping algorithm becomes a tremendously efficient tool when $c \gg 1$. Indeed, Kassam and Trefethen [32] illustrate that an entire order or magnitude in step-size can be gained for higher-order partial differential equations such as the Kuramoto–Sivashinky equation. In this case, the high c value is effectively created by linear, four-order diffusion. The order of magnitude in increased step-size Δt makes the implementation of the scheme above a must.

As a final comment, one can easily proceed with other schemes of this type. All that is needed is the starting point of the definition of the derivative or the fundamental theorem of calculus. Approximations are generated from there. The exponential scheme outlined above clearly takes advantage of the linear term by folding it into the integrating factor. All speciality stepping schemes typically do something of this form, or utilize a semi-implicit method, to maximize time-stepping performance and error reduction.

8.4 Boundary Value Problems: The Shooting Method

To this point, we have only considered the solutions of differential equations for which the initial conditions are known. However, many physical applications do not have specified initial conditions, but rather some given boundary (constraint) conditions. A simple example of such a problem is the second-order boundary value problem

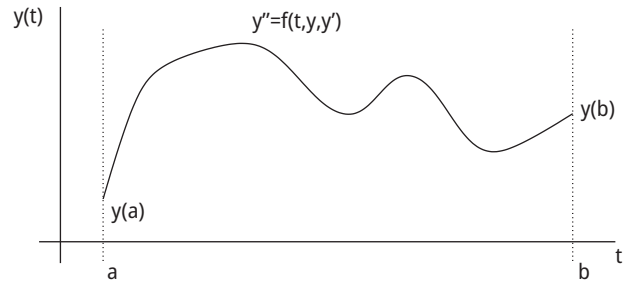


Figure 8.4 Graphical depiction of the structure of a typical solution to a boundary value problem with constraints at $t = a$ and $t = b$.

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{8.4.1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{8.4.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2. \tag{8.4.2b}$$

Thus, the solution is defined over a specific interval and must satisfy the relations (8.4.2a) at the end points of the interval. Figure 8.4 gives a graphical representation of a generic boundary value problem solution. We discuss the algorithm necessary to make use of the time-stepping schemes in order to solve such a problem.

8.4.1 The shooting method

The boundary value problems constructed here require information at the present time ($t = a$) and a future time ($t = b$). However, the time-stepping schemes developed previously only require information about the starting time $t = a$. Some effort is then needed to reconcile the time-stepping schemes with the boundary value problems presented here.

We begin by reconsidering the generic boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{8.4.3}$$

on $t \in [a, b]$ with the boundary conditions

$$y(a) = \alpha \tag{8.4.4a}$$

$$y(b) = \beta. \tag{8.4.4b}$$

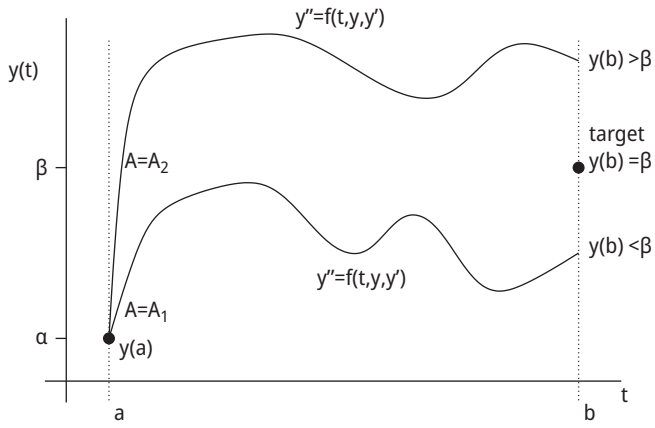


Figure 8.5 Solutions to the boundary value problem with $y(a) = \alpha$ and $y'(a) = A$. Here, two values of A are used to illustrate the solution behaviour and its lack of matching the correct boundary value $y(b) = \beta$. However, the two solutions suggest that a bisection scheme could be used to find the correct solution and value of A.

The stepping schemes considered thus far for second-order differential equations involve a choice of the initial conditions $y(a)$ and $y'(a)$. We can still approach the boundary value problem from this framework by choosing the "initial" conditions

$$y(a) = \alpha \tag{8.4.5a}$$

$$\frac{dy(a)}{dt} = A, \tag{8.4.5b}$$

where the constant A is chosen so that as we advance the solution to $t = b$ we find $y(b) = \beta$. The shooting method gives an iterative procedure with which we can determine this constant A. Figure 8.5 illustrates the solution of the boundary value problem given two distinct values of A. In this case, the value of $A = A_1$ gives a value for the initial slope which is too low to satisfy the boundary conditions (8.4.4a), whereas the value of $A = A_2$ is too large to satisfy (8.4.4a).

8.4.2 Computational algorithm

The above example demonstrates that adjusting the value of A in (8.4.5b) can lead to a solution which satisfies (8.4.4b). We can solve this using a self-consistent algorithm to search for the appropriate value of A which satisfies the original problem. The basic algorithm is as follows:

(1) Solve the differential equation using a time-stepping scheme with the initial conditions $y(a) = \alpha$ and $y'(a) = A$.

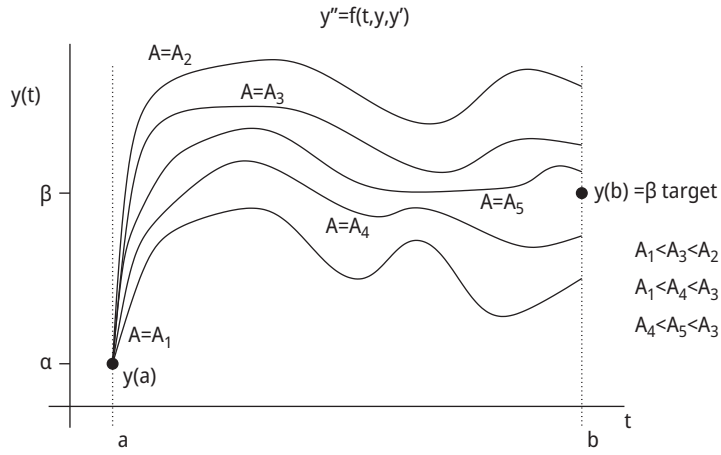


Figure 8.6 Graphical illustration of the shooting process which uses a bisection scheme to converge to the appropriate value of A for which $y(b) = \beta$.

(2) Evaluate the solution $y(b)$ at $t = b$ and compare this value with the target value of $y(b) = \beta$.

(3) Adjust the value of A (either bigger or smaller) until a desired level of tolerance and accuracy is achieved. A bisection method for determining values of A, for instance, may be appropriate.

(4) Once the specified accuracy has been achieved, the numerical solution is complete and is accurate to the level of the tolerance chosen and the discretization scheme used in the time-stepping.

We illustrate graphically a bisection process in Fig. 8.6 and show the convergence of the method to the numerical solution which satisfies the original boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. This process can occur quickly so that convergence is achieved in a relatively low number of iterations provided the differential equation is well behaved.

8.4.3 Shooting example

To illustrate the implementation of the shooting method, consider the following boundary value problem:

$$y'' + (x^2 - \sin x)y' - (\cos^2 x)y = 5 \qquad x \in [0, 1] \tag{8.4.6}$$

with the boundary conditions

$$y(0) = 3 \tag{8.4.7a}$$

$$y'(1) = 5. \tag{8.4.7b}$$

The first step is to write the above boundary value problem as an equivalent system of equations by defining $y_1 = y$ and $y_2 = y'$. This yields

$$y_1' = y_2 \tag{8.4.8a}$$

$$y_2' = -(x^2 - \sin x)y_2 + (\cos^2 x)y_1 + 5 \tag{8.4.8b}$$

$$y_1(0) = 3 \quad y_2(1) = 5\,. \tag{8.4.8c}$$

The idea is to replace the second boundary condition above, $y_2(1) = 5$, with $y_1'(0) = A$ where A is to be determined in the shooting algorithm. From exploring the differential equations with different values of A, it is found that $A > -3$ in order for the derivative at $x = 1$ to go from below the value of 5 to above the value of 5. The following code finds the solution in a fairly straightforward manner.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def bvpexam_rhs(y, x):
    return [y[1], -(x**2 - np.sin(x)) * y[1] + np.cos(x)**2 * y[0] + 5]

xspan = [0, 1]  # x range
A = -3  # initial derivative value
dA = 0.5  # step size for derivative adjustment

for j in range(100):
    y0 = [3, A]  # initial condition
    x = np.linspace(xspan[0], xspan[1], 100)  # grid for odeint
    ysol = odeint(bvpexam_rhs, y0, x)  # solve ODE

    if abs(ysol[-1, 1] - 5) < 10**(-6):  # check convergence
        break

    if ysol[-1, 1] < 5:  # adjust launch angle
        A += dA  # if below five, make A bigger
    else:
        A -= dA  # if above five, make A smaller
        dA /= 2  # refine search now
```

Figure 8.7 depicts the final solution $y(x) = y_1(x)$ along with the derivative $y'(x) = y_2(x)$. The dotted line is the initial guess ($A = -3$) for a solution and the starting point of the shooting

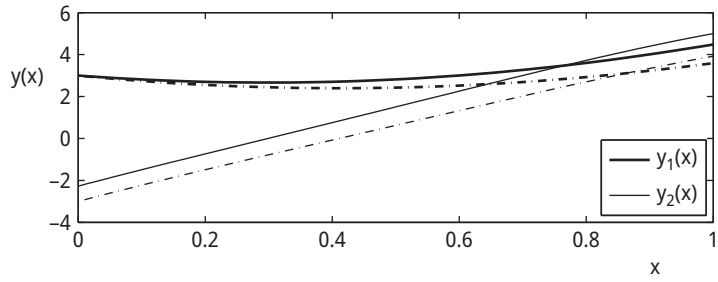


Figure 8.7 Solution of the boundary value problem depicting the solution $y(x) = y_1(x)$ (bolded lines) and its derivative $y'(x) = y_2(x)$. The dotted lines are the initial guess of the shooting algorithm for which $y'(1) = y_2(1) = A = -3$.

algorithm. Note that the algorithm uses something like a bisection algorithm for refining the search for the appropriate value of A.

8.4.4 Eigenvalues and eigenfunctions: The infinite domain

Boundary value problems often arise as eigenvalue systems for which the eigenvalue and eigenfunction must both be determined. As an example of such a problem, we consider the second-order differential equation on the infinite line

$$\frac{d^2\psi_n}{dx^2} + [n(x) - \beta_n]\,\psi_n = 0 \tag{8.4.9}$$

with the boundary conditions $\psi_n(x) \to 0$ as $x \to \pm\infty$. For this example, we consider the spatial function $n(x)$ which is given by

$$n(x) = n_0 \begin{cases} 1 - |x|^2 & 0 \le |x| \le 1 \\ 0 & |x| > 1 \end{cases} \tag{8.4.10}$$

with n_0 being an arbitrary constant. Figure 8.8 shows the spatial dependence of $n(x)$. The parameter β_n in this problem is the eigenvalue. For each eigenvalue, we can calculate a normalized eigenfunction ψ_n. The standard normalization requires $\int_{-\infty}^{\infty} |\psi_n|^2 dx = 1$.

Although the boundary conditions are imposed as $x \to \pm\infty$, computationally we require a finite domain. We thus define our computational domain to be $x \in [-L, L]$ where $L \gg 1$. Since $n(x) = 0$ for $|x| > 1$, the governing equation reduces to

$$\frac{d^2\psi_n}{dx^2} - \beta_n\psi_n = 0 \quad |x| > 1 \tag{8.4.11}$$

which has the general solution

$$\psi_n = c_1 \exp\left(\sqrt{\beta_n}x\right) + c_2 \exp\left(-\sqrt{\beta_n}x\right) \tag{8.4.12}$$

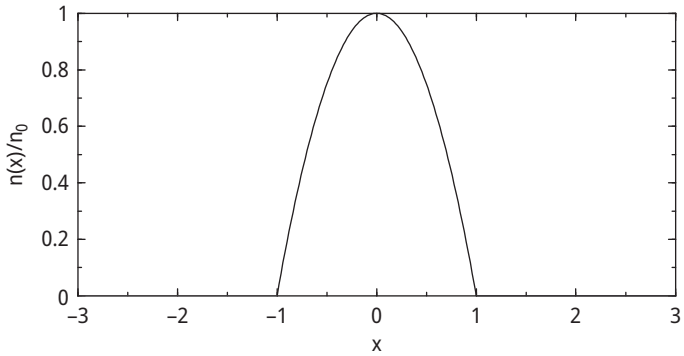


Figure 8.8 Plot of the spatial function $n(x)$.

for $\beta_n \geq 0$. Note that we can only consider values of $\beta_n \geq 0$ since for $\beta_n < 0$, the general solution becomes $\psi_n = c_1 \cos(\sqrt{|\beta_n|}x) + c_2 \sin(-\sqrt{|\beta_n|}x)$ which does not decay to zero as $x \to \pm\infty$. In order to ensure that the decay boundary conditions are satisfied, we must eliminate one of the two linearly independent solutions of the general solution. In particular, we must have

$$x \to \infty : \qquad \psi_n = c_2 \exp\left(-\sqrt{\beta_n}x\right) \tag{8.4.13a}$$

$$x \to -\infty : \quad \psi_n = c_1 \exp\left(\sqrt{\beta_n}x\right). \tag{8.4.13b}$$

Thus, the requirement that the solution decays at infinity eliminates one of the two linearly independent solutions. Alternatively, we could think of this situation as being a case where only one linearly independent solution is allowed as $x \to \pm\infty$. But a single linearly independent solution corresponds to a first-order differential equation. Therefore, the decay solutions (8.4.13b) can equivalently be thought of as solutions to the following first-order equations:

$$x \to \infty : \qquad \frac{d\psi_n}{dx} + \sqrt{\beta_n}\psi_n = 0 \tag{8.4.14a}$$

$$x \to -\infty : \quad \frac{d\psi_n}{dx} - \sqrt{\beta_n}\psi_n = 0. \tag{8.4.14b}$$

From a computational viewpoint then, the effective boundary conditions to be considered on the computational domain $x \in [-L, L]$ are the following:

$$x = L : \qquad \frac{d\psi_n(L)}{dx} = -\sqrt{\beta_n}\psi_n(L) \tag{8.4.15a}$$

$$x = -L : \quad \frac{d\psi_n(-L)}{dx} = \sqrt{\beta_n}\psi_n(-L). \tag{8.4.15b}$$

In order to solve the problem, we write the governing differential equation as a system of equations. Thus, we let $x_1 = \psi_n$ and $x_2 = d\psi_n/dx$, which gives

$$x_1' = \psi_n' = x_2 \tag{8.4.16a}$$

$$x_2' = \psi_n'' = [\beta_n - n(x)]\,\psi_n = [\beta_n - n(x)]\,x_1. \tag{8.4.16b}$$

In matrix form, we can write the governing system as

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{8.4.17}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions (8.4.15b) are

$$x = L: \qquad x_2 = -\sqrt{\beta_n} x_1 \tag{8.4.18a}$$

$$x = -L: \qquad x_2 = \sqrt{\beta_n} x_1. \tag{8.4.18b}$$

The formulation of the boundary value problem is thus complete. It remains to develop an algorithm to find the eigenvalues β_n and corresponding eigenfunctions ψ_n. Figure 8.9

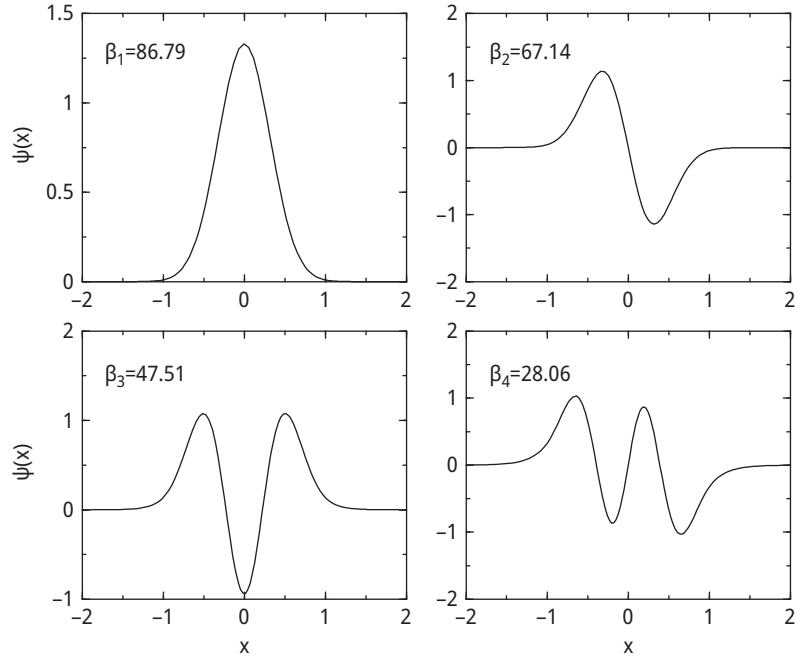


Figure 8.9 Plot of the first four eigenfunctions along with their eigenvalues β_n. For this example, $L = 2$ and $n_0 = 100$. These eigenmode structures are typical of those found in quantum mechanics and electromagnetic waveguides.

illustrates the first four eigenfunctions and their associated eigenvalues for $n_0 = 100$ and $L = 2$.

8.5 Implementation of Shooting and Convergence Studies

The implementation of the shooting scheme relies on the effective use of a time-stepping algorithm along with a root-finding method for choosing the appropriate initial conditions that solve the boundary value problem. The specific system to be considered is similar to that developed in the last section. We consider

$$\mathbf{x}' = \begin{pmatrix} 0 & 1 \\ \beta_n - n(x) & 0 \end{pmatrix} \mathbf{x} \tag{8.5.1}$$

where $\mathbf{x} = (x_1 \;\; x_2)^T = (\psi_n \;\; d\psi_n/dx)^T$. The boundary conditions are simplified in this case to be

$$x = 1: \qquad \psi_n(1) = x_1(1) = 0 \tag{8.5.2a}$$

$$x = -1: \qquad \psi_n(-1) = x_1(-1) = 0. \tag{8.5.2b}$$

At this stage, we will also assume that $n(x) = n_0$ for simplicity.

With the problem thus defined, we turn our attention to the key aspects in the computational implementation of the boundary value problem solver. These are

- **FOR** loops
- **IF** statements
- time-stepping algorithms: **ode23, ode45, ode113, ode15s**
- step-size control
- code development and flow.

Every code will be controlled by a set of FOR loops and IF statements. It is imperative to have proper placement of these control statements in order for the code to operate successfully.

8.5.1 Convergence

In addition to developing a successful code, it is reasonable to ask whether your numerical solution is actually correct. Thus far, the premise has been that discretization should provide an accurate approximation to the true solution provided the time-step Δt is small enough. Although in general this philosophy is correct, every numerical algorithm should

be carefully checked to determine if it indeed converges to the true solution. The time-stepping schemes considered previously already hint at how the solutions should converge: fourth-order Runge–Kutta converges like Δt^4, second-order Runge–Kutta converges like Δt^2, and second-order predictor–corrector schemes converge like Δt^2. Thus the algorithm for checking convergence is as follows:

(1) Solve the differential equation using a time-step $\Delta t*$ which is very small. This solution will be considered the exact solution. Recall that we would in general like to take Δt as large as possible for efficiency purposes.

(2) Using a much larger time-step Δt, solve the differential equation and compare the numerical solution with that generated from $\Delta t*$. Cut this time-step in half and compare once again. In fact, continue cutting the time-step in half: $\Delta t, \Delta t/2, \Delta t/4, \Delta t/8, \ldots$ in order to compare the difference in the exact solution to this hierarchy of solutions.

(3) The difference between any run $\Delta t*$ and Δt is considered the error. Although there are many definitions of error, a practical error measurement is the root-mean square error $E = \left[(1/N)\sum_{i=1}^{N} |y_{\Delta t*} - y_{\Delta t}|^2\right]^{1/2}$. Once calculated, it is possible to verify the convergence law of Δt^2, for instance, with a second-order Runge–Kutta.

8.5.2 Flow control

In order to begin coding, it is always prudent to construct the basic structure of the algorithm. In particular, it is good to determine the number of FOR loops and IF statements which may be required for the computations. What is especially important is determining the hierarchic structure for the loops. To solve the boundary value problem proposed here, we require two FOR loops and one IF statement block. The outermost FOR loop of the code should determine the number of eigenvalues and eigenmodes to be searched for. Within this FOR loop, there exists a second FOR loop which iterates the shooting method so that the solution converges to the correct boundary value solution. This second FOR loop has a logical IF statement which needs to check whether the solution has indeed converged to the boundary value solution, or whether adjustment of the value of β_n is necessary and the iteration procedure needs to be continued. Figure 8.10 illustrates the backbone of the numerical code for solving the boundary value problem. It includes the two FOR loops and logical IF statement block as the core of its algorithmic structure. For a nonlinear problem, a third FOR loop would be required for A in order to achieve the normalization of the eigenfunctions to unity.

The various pieces of the code are constructed here using the Python programming language. We begin with the initialization of the parameters.

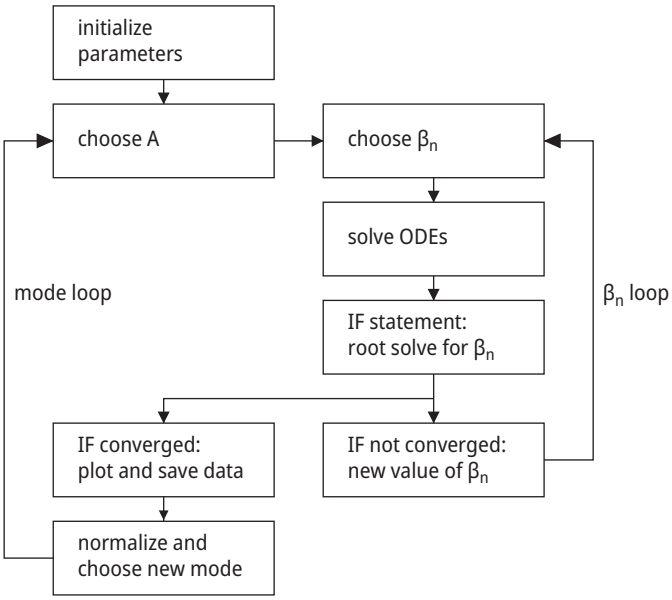


Figure 8.10 Basic algorithm structure for solving the boundary value problem. Two FOR loops are required to step through the values of β_n and A along with a single IF statement block to check for convergence of the solution.

Initialization

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

tol = 1e-4  # define a tolerance level
col = ['r', 'b', 'g', 'c', 'm', 'k']  # eigenfunc colors
n0 = 100; A = 1; x0 = [0, A]; xp = [-1, 1]
xshoot =  np.linspace(xp[0], xp[1],1000)
```

Upon completion of the initialization process for the parameters which are not involved in the main loop of the code, we move into the main FOR loop which searches out a specified number of eigenmodes. Embedded in this FOR loop is a second FOR loop which attempts different values of β_n until the correct eigenvalue is found. An IF statement is used to check the convergence of values of β_n to the appropriate value.

Main program

```
def shoot2(x, dummy, n0, beta):
  return [x[1], (beta - n0) * x[0]]

beta_start = n0  # beginning value of beta
for modes in range(1, 6):  # begin mode loop
  beta = beta_start  # initial value of eigenvalue beta
  dbeta = n0 / 100  # default step size in beta
  for _ in range(1000):  # begin convergence loop for beta
        y = odeint(shoot2, x0, xshoot, args=(n0,beta))

        if abs(y[-1, 0] - 0) < tol:  # check for convergence
            print(beta)  # write out eigenvalue
            break  # get out of convergence loop

        if (-1) ** (modes + 1) * y[-1, 0] > 0:
            beta -= dbeta
        else:
            beta += dbeta / 2
            dbeta /= 2

  beta_start = beta - 0.1 # after finding eigenvalue, pick new start
  norm = np.trapz(y[:, 0] * y[:, 0], xshoot) # calculate
       the normalization
  plt.plot(xshoot, y[:, 0] / np.sqrt(norm), col[modes - 1]) # plot
    modes
```

The code uses *ode45*, which is a fourth-order Runge–Kutta method, to solve the differential equation and advance the solution. The function *shoot2.m* is called in this routine.

This code will find the first five eigenvalues and plot their corresponding normalized eigenfunctions. The bisection method implemented to adjust the values of β_n to find the boundary value solution is based upon observations of the structure of the even and odd eigenmodes. In general, it is always a good idea to first explore the behaviour of the solutions of the boundary value problem before writing the shooting routine. This will give important insights into the behaviour of the solutions and will allow for a proper construction of an accurate and efficient bisection method. Figure 8.11 illustrates several characteristic features of this boundary value problem. In Figs. 8.11(a) and 8.11(b), the behaviour of the solution near the first even and first odd solution is exhibited. From Fig. 8.11(a), it is seen that for the even modes, increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. This observation

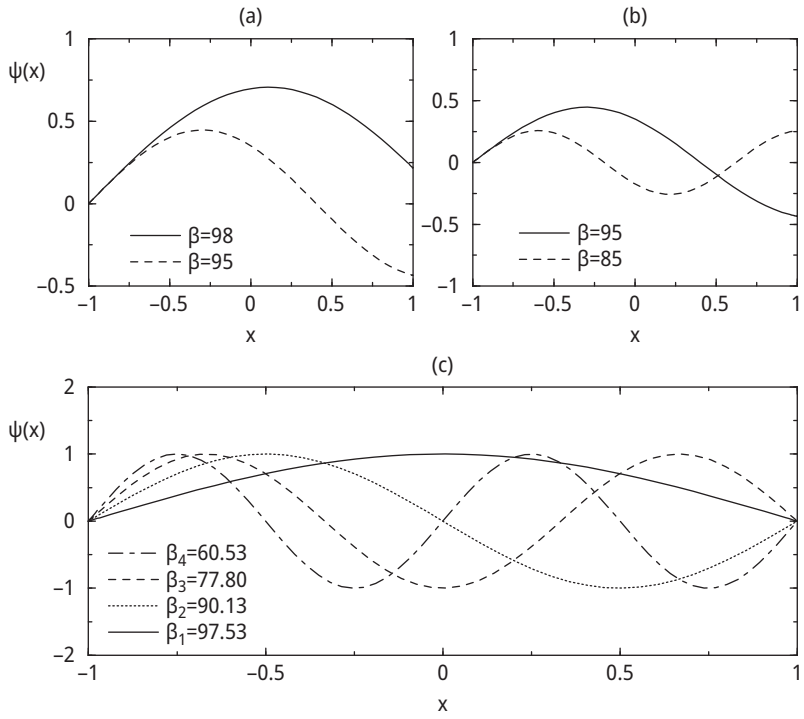


Figure 8.11 In (a) and (b), the behaviour of the solution near the first even and first odd solution is depicted. Note that for the even modes increasing values of β bring the solution from $\psi_n(1) > 0$ to $\psi_n(1) < 0$. In contrast, odd modes go from $\psi_n(1) < 0$ to $\psi_n(1) > 0$ as β is increased. In (c), the first four normalized eigenmodes along with their corresponding eigenvalues are illustrated for $n_0 = 100$.

forms the basis for the bisection method developed in the code. Figure 8.11(c) illustrates the first four normalized eigenmodes along with their corresponding eigenvalues.

8.6 Boundary Value Problems: Direct Solve and Relaxation

The shooting method is not the only method for solving boundary value problems. The direct method of solution relies on Taylor expanding the differential equation itself. For linear problems, this results in a matrix problem of the form $\mathbf{Ax} = \mathbf{b}$. For nonlinear problems, a nonlinear system of equations must be solved using a relaxation scheme, i.e. a Newton or secant method. The prototypical example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \tag{8.6.1}$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \tag{8.6.2a}$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2. \tag{8.6.2b}$$

Thus, the solution is defined over a specific interval and must satisfy the relations (8.6.2a) at the end points of the interval.

Before considering the general case, we simplify the method by considering the linear boundary value problem

$$\frac{d^2y}{dt^2} = p(t)\frac{dy}{dt} + q(t)y + r(t) \tag{8.6.3}$$

on $t \in [a, b]$ with the simplified boundary conditions

$$y(a) = \alpha \tag{8.6.4a}$$

$$y(b) = \beta. \tag{8.6.4b}$$

Taylor expanding the differential equation and boundary conditions will generate the linear system of equations which solve the boundary value problem.

To see how the Taylor expansions are useful, consider the following two Taylor series:

$$f(t + \Delta t) = f(t) + \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} + \frac{\Delta t^3}{3!}\frac{d^3f(c_1)}{dt^3} \tag{8.6.5a}$$

$$f(t - \Delta t) = f(t) - \Delta t \frac{df(t)}{dt} + \frac{\Delta t^2}{2!}\frac{d^2f(t)}{dt^2} - \frac{\Delta t^3}{3!}\frac{d^3f(c_2)}{dt^3} \tag{8.6.5b}$$

where $c_1 \in [t, t+\Delta t]$ and $c_2 \in [t, t-\Delta t]$. Subtracting these two expressions gives

$$f(t + \Delta t) - f(t - \Delta t) = 2\Delta t \frac{df(t)}{dt} + \frac{\Delta t^3}{3!}\left(\frac{d^3f(c_1)}{dt^3} + \frac{d^3f(c_2)}{dt^3}\right). \tag{8.6.6}$$

By using the mean value theorem of calculus, we find $f'''(c) = (f'''(c_1) + f'''(c_2))/2$. Upon dividing the above expression by $2\Delta t$ and rearranging, we find the following expression for the first derivative:

$$\frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} - \frac{\Delta t^2}{6}\frac{d^3f(c)}{dt^3} \tag{8.6.7}$$

where the last term is the truncation error associated with the approximation of the first derivative using this particular Taylor series-generated expression. Note that the truncation error in this case is $O(\Delta t^2)$. We could improve on this by continuing our Taylor expansion and truncating it at higher orders in Δt. This would lead to higher accuracy schemes. Further, we could also approximate the second, third, fourth, and higher derivatives using this

technique. It is also possible to generate backward and forward difference schemes by using points only behind or in front of the current point, respectively. Tables 3.1–3.3 summarize the second-order and fourth-order central difference schemes along with the forward- and backward-difference formulas which are accurate to second order.

To solve the simplified linear boundary value problem above, which is accurate to second order, we use Table 1 for the second and first derivatives. The boundary value problem then becomes

$$\frac{y(t+\Delta t)-2y(t)+y(t-\Delta t)}{\Delta t^2} = p(t)\frac{y(t+\Delta t)-y(t-\Delta t)}{2\Delta t}+q(t)y(t)+r(t) \tag{8.6.8}$$

with the boundary conditions $y(a)=\alpha$ and $y(b)=\beta$. We can rearrange this expression to read

$$\left[1-\frac{\Delta t}{2}p(t)\right]y(t+\Delta t)-\left[2+\Delta t^2 q(t)\right]y(t)+\left[1+\frac{\Delta t}{2}p(t)\right]y(t-\Delta t)=\Delta t^2 r(t). \tag{8.6.9}$$

We discretize the computational domain and denote $t_0=a$ to be the left boundary point and $t_N=b$ to be the right boundary point. This gives the boundary conditions

$$y(t_0)=y(a)=\alpha \tag{8.6.10a}$$

$$y(t_N)=y(b)=\beta. \tag{8.6.10b}$$

The remaining $N-1$ points can be recast as a matrix problem $\mathbf{Ax}=\mathbf{b}$ where

$$\mathbf{A}=\begin{bmatrix} 2+\Delta t^2 q(t_1) & -1+\frac{\Delta t}{2}p(t_1) & 0 & \cdots & & 0 \\ -1-\frac{\Delta t}{2}p(t_2) & 2+\Delta t^2 q(t_2) & -1+\frac{\Delta t}{2}p(t_2) & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & & & \ddots & \ddots & -1+\frac{\Delta t}{2}p(t_{N-2}) \\ 0 & \cdots & & 0 & -1-\frac{\Delta t}{2}p(t_{N-1}) & 2+\Delta t^2 q(t_{N-1}) \end{bmatrix} \tag{8.6.11}$$

and

$$\mathbf{x}=\begin{bmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_{N-2}) \\ y(t_{N-1}) \end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix} -\Delta t^2 r(t_1)+(1+\Delta t p(t_1)/2)y(t_0) \\ -\Delta t^2 r(t_2) \\ \vdots \\ -\Delta t^2 r(t_{N-2}) \\ -\Delta t^2 r(t_{N-1})+(1-\Delta t p(t_{N-1})/2)y(t_N) \end{bmatrix}. \tag{8.6.12}$$

Thus, the solution can be found by a direct solve of the linear system of equations.

8.6.1 Nonlinear systems

A similar solution procedure can be carried out for nonlinear systems. However, difficulties arise from solving the resulting set of nonlinear algebraic equations. We can once again consider the general differential equation and expand with second-order accurate schemes:

$$y'' = f(t, y, y') \rightarrow \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2} = f\left(t, y(t), \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}\right). \quad (8.6.13)$$

We discretize the computational domain and denote $t_0 = a$ to be the left boundary point and $t_N = b$ to be the right boundary point. Considering again the simplified boundary conditions $y(t_0) = y(a) = \alpha$ and $y(t_N) = y(b) = \beta$ gives the following nonlinear system for the remaining $N - 1$ points.

$$\begin{aligned}
&2y_1 - y_2 - \alpha + \Delta t^2 f(t_1, y_1, (y_2 - \alpha)/2\Delta t) = 0 \\
&-y_1 + 2y_2 - y_3 + \Delta t^2 f(t_2, y_2, (y_3 - y_1)/2\Delta t) = 0 \\
&\qquad\qquad \vdots \\
&-y_{N-3} + 2y_{N-2} - y_{N-1} + \Delta t^2 f(t_{N-2}, y_{N-2}, (y_{N-1} - y_{N-3})/2\Delta t) = 0 \\
&-y_{N-2} + 2y_{N-1} - \beta + \Delta t^2 f(t_{N-1}, y_{N-1}, (\beta - y_{N-2})/2\Delta t) = 0.
\end{aligned}$$

This $(N-1) \times (N-1)$ nonlinear system of equations can be very difficult to solve and imposes a severe constraint on the usefulness of the scheme. However, there may be no other way of solving the problem and a solution to this system of equations must be computed. Further complicating the issue is the fact that for nonlinear systems such as these, there are no guarantees about the existence or uniqueness of solutions. The best approach is to use a *relaxation* scheme that is based upon Newton or secant method iterations.

8.7 **Implementing Python for Boundary Value Problems**

Both a shooting technique and a direct discretization method have been developed here for solving boundary value problems. More generally, one would like to use a high-order method that is robust and capable of solving general, nonlinear boundary value problems. Python provides a convenient and easy-to-use routine that is capable of solving fairly sophisticated problems. The algorithm relies on an iteration structure for solving nonlinear systems of equations. In particular, the code is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula, and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in $x \in [a, b]$. Mesh selection and error control are based on the residual of the continuous solution. Since it is an iteration scheme, its effectiveness will ultimately rely on your ability to provide the algorithm with an initial guess for the solution.

Two example codes will be demonstrated here. The first is a simple linear, constant coefficient second-order equation with fairly standard boundary conditions. The second example is nonlinear with an undetermined parameter (eigenvalue) that must also be determined. Both illustrate the power and ease of use of the built-in boundary value solver of Python.

8.7.1 Linear boundary value problem

As a simple and particular example of a boundary value problem, consider the following:

$$y'' + 3y' + 6y = 5 \tag{8.7.1}$$

on the domain $x \in [1, 3]$ and with boundary conditions

$$y(1) = 3 \tag{8.7.2a}$$

$$y(3) + 2y'(3) = 5. \tag{8.7.2b}$$

This problem can be solved in a fairly straightforward manner using analytic techniques. However, we will pursue here a numerical solution instead.

As with any differential equation solver, the equation must first be put into the form of a system of first-order equations. Thus, by introducing the variables

$$y_1 = y(x) \tag{8.7.3a}$$

$$y_2 = y'(x) \tag{8.7.3b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{8.7.4a}$$

$$y_2' = 5 - 3y_2 - 6y_1 \tag{8.7.4b}$$

with the transformed boundary conditions

$$y_1(1) = 3 \tag{8.7.5a}$$

$$y_1(3) + 2y_2(3) = 5. \tag{8.7.5b}$$

In order to implement the boundary value problem in Python, the boundary conditions need to be placed in the general form

$$f(y_1, y_2) = 0 \text{ at } x = x_L \tag{8.7.6a}$$

$$g(y_1, y_2) = 0 \text{ at } x = x_R \tag{8.7.6b}$$

where $f(y_1, y_2)$ and $g(y_1, y_2)$ are the boundary value functions at the left (x_L) and right (x_R) boundary points. This then allows us to rewrite the boundary conditions in (8.7.5b) as the following:

$$f(y_1, y_2) = y_1 - 3 = 0 \qquad \text{at } x_L = 1 \tag{8.7.7a}$$

$$g(y_1, y_2) = y_1 + 2y_2 - 5 = 0 \text{ at } x_R = 3\,. \tag{8.7.7b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (8.7.4b) and its boundary conditions (8.7.7b).

The boundary value solver requires three pieces of information: the equation to be solved, its associated boundary conditions, and your initial guess for the solution. The first two lines of the following code performs all three of these functions:

```
from scipy.integrate import solve_bvp
import matplotlib.pyplot as plt

def bvp_rhs(x, y):
    return np.vstack((y[1], 5 - 3 * y[1] - 6 * y[0]))

def bvp_bc(ya, yb):
    return np.array([ya[0] - 3, yb[0] + 2 * yb[1] - 5])

# Define initial mesh and guess
x_init = np.linspace(1, 3, 10)
y_init = np.zeros((2, x_init.size))

# Solve the boundary value problem
sol = solve_bvp(bvp_rhs, bvp_bc, x_init, y_init)

# Evaluate the solution on a finer mesh
x_eval = np.linspace(1, 3, 100)
BS = sol.sol(x_eval)
plt.plot(x_eval, BS[0])
```

We dissect this code by first considering the first line of code for generating a Python data structure for use as the initial data. In this initial line of code, the **linspace** command is used to define the initial mesh that goes from $x = 1$ to $x = 3$ with 10 equally spaced points. In this example, the initial values of $y_1(x) = y(x)$ and $y_2(x) = y'(x)$ are zero. If you have no guess, then this is probably your best guess to start with. Thus, you not only guess the initial guess, but the initial grid on which to find the solution.

Once the guess and initial mesh is generated, two functions are called with the **@bvp_rhs** and **@bvp_bc** function calls. These are functions representing the differential equation and boundary conditions (8.7.4b) and (8.7.7b), Note that what is passed into the boundary condition function is the value of the vector $\mathbf{y}$ at the left ($\mathbf{yL}$) and right ($\mathbf{yR}$) of the computational domain, i.e. at the values of $x = 1$ and $x = 3$.

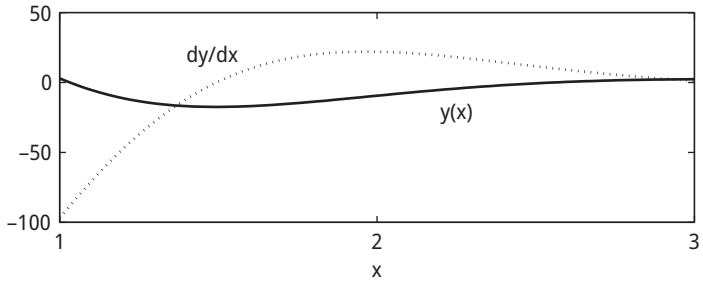


Figure 8.12 The solution of the boundary value problem (8.7.4b) with (8.7.7b). The solid line is $y(x)$, while the dotted line is its derivative $dy(x)/dx$.

What is produced by the boundary value solver is a data structure **sol**. This data structure contains a variety of information about the problem, including the solution $y(x)$. To extract the solution, the final two lines of code in the main program are used. Specifically, the command evaluates the solution at the points specified by the vector **x**, which in this case is a linear space of 100 points between 1 and 3. The solution can then simply be plotted once the values of $y(x)$ have been extracted. Figure 8.12 shows the solution and its derivative that are produced from the boundary value solver.

8.7.2 Nonlinear eigenvalue problem

As a more sophisticated example of a boundary value problem, we will consider a fully nonlinear eigenvalue problem. Thus, consider the following:

$$y'' + (100 - \beta)y + \gamma y^3 = 0 \tag{8.7.8}$$

on the domain $x \in [-1, 1]$ and with boundary conditions

$$y(-1) = 0 \tag{8.7.9a}$$

$$y(1) = 0. \tag{8.7.9b}$$

This problem cannot be solved using analytic techniques due to the complexity introduced by the nonlinearity. But a numerical solution can be fairly easily constructed. Note the similarity between this problem and that considered in the shooting section.

As before, the equation must first be put into the form of a system of first-order equations. Thus, by introducing the variables

$$y_1 = y(x) \tag{8.7.10a}$$

$$y_2 = y'(x) \tag{8.7.10b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{8.7.11a}$$

$$y_2' = (\beta - 100)y_1 - \gamma y_1^3 \tag{8.7.11b}$$

with the transformed boundary conditions

$$y_1(-1) = 0 \tag{8.7.12a}$$

$$y_1(1) = 0. \tag{8.7.12b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (11b) and its boundary conditions (8.7.12b). Note that unlike before, we do not know the parameter β. Thus, it must be determined along with the solution. As with the initial conditions, we will also guess an initial value of β and let the solver converge to the appropriate value of β. Note that for such nonlinear problems, the effectiveness of the method relies almost exclusively on providing a good initial guess.

As before, the boundary value solver requires three pieces of information: the equation to be solved, its associated boundary conditions, and your initial guess for the solution and the parameter β. The first line of the code gives the initial guess for β, while the next two lines of the following code perform the three remaining functions:

```
import numpy as np
from scipy.integrate import solve_bvp
import matplotlib.pyplot as plt

def bvp_rhs2(x, y, beta):
    return np.vstack((y[1], (beta - 100) * y[0] - y[0]**3))

def bvp_bc2(yl, yr, beta):
    return np.array([yl[0], yl[1] - 0.1, yr[0]])

def mat4init(x):
    return np.array([np.cos((np.pi/2)*x),-(np.pi / 2)
        *np.sin((np.pi/2)*x)])

beta = 99
x_init = np.linspace(-1, 1, 50)
init2 = solve_bvp(bvp_rhs2, bvp_bc2, x_init, mat4init(x_init),
     p=[beta])

x2 = np.linspace(-1, 1, 100)
BS2 = init2.sol(x2)
plt.plot(x2, BS2[0])
```

In this case, there are now three function calls: **@bvp_rhs2**, **@bvp_bc2**, and **@mat4init**. This calls the differential equation, its boundary values and its initial guess, respectively.

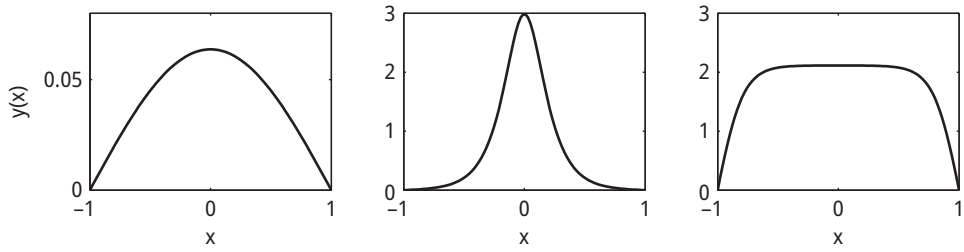


Figure 8.13 Three different nonlinear eigenvalue solutions to (11b) and its boundary conditions (8.7.12b). The left panel has the constraint condition $y'(-1) = 0.1$ with $\gamma = 1$, the middle panel has $y'(-1) = 0.1$ with $\gamma = 10$, and the right panel has $y'(-1) = 10$ with $\gamma = -10$. Such solutions are called *nonlinear ground states* of the eigenvalue problem.

We begin with the initial guess. Our implementation example of the shooting method showed that the first linear solution behaved like a cosine function. Thus, we can guess that $y_1(x) = y(x) = \cos[(\pi/2)x]$, where the factor of $\pi/2$ is chosen to make the solution zero at $x = \pm 1$. Note that initially 50 points are chosen between $x = -1$ and $x = 1$.

Again, it must be emphasized that the success of the solver relies almost exclusively on the above subroutine and guess. Without a good guess, especially for nonlinear problems, you may find a solution, but just not the one you want.

The equation itself is handled in the subroutine **bvp_rhs2.m**.

Finally, the implementation of the boundary conditions, or constraints, must be imposed. There is something very important to note here: we started with two constraints since we have a second-order differential equation, i.e. $y(\pm 1) = 0$. However, since we do not know the value of β, the system is currently an underdetermined system of equations. In order to remedy this, we need to impose one more constraint on the system. This is somewhat arbitrary unless there is a natural constraint for you to choose in the system. Here, we will simply choose $dy(-1)/dx = 0.1$, i.e. we will impose the launch angle at the left.

With these three constraints, the iteration routine not only finds the solution but also the appropriate value of β that satisfies the above constraints.

Executing the routine and subroutines generates the solution to the boundary value problem. One can also experiment with different guesses and values of β to generate new and interesting solutions. Three such solutions are demonstrated in Fig. 8.13 as a function of the β value and the guess angle $dy(-1)/dx$.

8.8 Linear Operators and Computing Spectra

As a final application of boundary value problems, we will consider the ability to accurately compute the spectrum of linear operators. Linear operators often arise in the context of evaluating the stability of solutions to partial differential equations. Indeed, stability plays a key role in many branches of science and engineering, including aspects of fluid mechanics, pattern formation with reaction–diffusion models, high-speed transmission of optical information, and the feasibility of magnetohydrodynamics fusion devices, to name a few.

If one can find solutions for a given particular differential equation, then the stability of that solution becomes critical in determining the ultimate behaviour of the system. Specifically, if a physical phenomenon is observable and persists, then the corresponding solution to a valid mathematical model should be stable. If, however, instability is established, the nature of the unstable modes suggests what patterns may develop from the unstable solutions. Finally, for many problems of physical interest, fundamental mathematical models are well established. However, in many cases, these fundamental models are too complicated to allow for detailed analysis, thus leading to the study of simpler approximate (linear) models using reductive perturbation methods.

To further illustrate these concepts, consider a generic partial differential equation of the form

$$\frac{\partial u}{\partial t} = N\left(x, u, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, \cdots\right) \tag{8.8.1}$$

where the function N is generically nonconstant coefficient in x and a nonlinear function of $u(x,t)$ and its derivatives. Here, we will assume that this function is well behaved so that solutions can exist by the basic Cauchy–Kovaleskaya theorem. In addition, there are boundary conditions associated with (8.8.1). Such boundary constraints may be imposed on either a finite or infinite domain, and an example of each case will be considered in what follows.

In what follows, only the stability of equilibrium solutions will be considered. Equilibrium solutions are found when $\partial u/\partial t = 0$. Thus, there would exist a time-independent solution $U(x)$ such that

$$N\left(x, U, \frac{\partial U}{\partial x}, \frac{\partial^2 U}{\partial x^2}, \cdots\right) = 0. \tag{8.8.2}$$

If one could find such a solution $U(x)$, then its stability could be computed. Again, if the solution is stable, then it may be observable in the physical system. If it is unstable, then it will not be observed in practice. However, the unstable modes of such a system may potentially give clues to the resulting behaviour of the system.

To generate a linear stability problem for the equilibrium solution $U(x)$ of (8.8.1), one would *linearize* about the solution so that

$$u(x,t) = U(x) + \epsilon v(x,t) \tag{8.8.3}$$

where the parameter $\epsilon \ll 1$ so that the function $v(x,t)$ is only a small perturbation from the equilibrium solution. If the function $v(x,t) \to \infty$ as $t \to \infty$, the system is said to be unstable. If $v(x,t) \to 0$ as $t \to \infty$, the system is said to be asymptotically stable. If $v(x,t)$ remains $O(1)$ as $t \to \infty$, the system is said to simply be stable. Thus, a determination of what happens to $v(x,t)$ is required. Plugging in (8.8.3) into (8.8.1) and Taylor expanding using the fact that $\epsilon \ll 1$ gives the equation

$$\frac{\partial v}{\partial t} = \mathcal{L}\left[U(x)\right] v + O(\epsilon) \tag{8.8.4}$$

where the $O(\epsilon)$ represents all the terms from the Taylor expansion that are $O(\epsilon)$ or smaller. Finally, by assuming that the function $v(x,t)$ takes the following form:

$$v(x,t) = w(x)\exp(\lambda t) \tag{8.8.5}$$

the following *linear* eigenvalue problem (spectral problem) results:

$$\mathcal{L}\left[U(x)\right]v = \lambda v \tag{8.8.6}$$

where $\mathcal{L}$ is the linear operator associated with the stability of the equilibrium solution $U(x)$. Note that by our definitions of stability above and the solution form (8.8.5), the equilibrium solution is unstable if any real part of the eigenvalue is positive: $\Re\{\lambda\} > 0$. Stability is established if $\Re\{\lambda\} \leq 0$, with asymptotic stability occurring if $\Re\{\lambda\} < 0$. Thus, computing the eigenvalues of the linear operator is the critical step in evaluating stability.

8.8.1 Sturm–Liouville theory

Of course, anybody familiar with Sturm–Liouville theory [29] will recognize the importance of computing the spectra of the linearized operators. For Sturm–Liouville problems, the linear operator and its associated eigenvalues take on the form

$$\mathcal{L}v = -\frac{d}{dx}\left[p(x)\frac{dv}{dx}\right] + q(x)v = \lambda w(x)v \tag{8.8.7}$$

where $p(x) > 0$, $q(x)$ and $w(x) > 0$ are specified on the interval $x \in [a,b]$. In the simplest case, these are continuous on the entire interval. The associated boundary conditions are given by

$$\alpha_1 v(a) + \alpha_2 \frac{dv(a)}{dx} = 0 \quad \text{and} \quad \beta_1 v(b) + \beta_2 \frac{dv(b)}{dx} = 0. \tag{8.8.8}$$

It is known that the Sturm–Liouville problem has some very nice properties, including the fact that the eigenvalues are all real and distinct. Many classic physical systems of interest naturally fall within Sturm–Liouville theory, thus propagating a variety of special functions such as Bessel functions, Legendre functions, Laguerre polynomials, etc. Indeed, much of special-function theory involves Sturm–Liouville theory at its core. In certain cases, the ideas of Sturm–Liouville can be extended to finding solutions to nonlinear eigenvalue problems [33].

8.8.2 Derivative operators

Although much is known in the literature about Sturm–Liouville theory, the theory often relates properties about the eigenfunctions and eigenvectors in the abstract. To compute the actual spectra of (8.8.7), we will advocate a numerical procedure based upon finite difference discretization. Shooting methods can also be applied, but they will not be considered

in what follows. What is immediately apparent from (8.8.7), and more generally (8.8.6), is the presence of derivative operators. From the finite difference perspective, these are no more than matrix operations on the discretized (vector) solution. Following the ideas of Section 3.3, we can construct a number of first- and second-derivative matrices that are continually used in practice. The domain $x \in [a,b]$ is discretized into $N+1$ intervals so that $x_0 = a$ and $x_{N+1} = b$.

To begin, we will consider constructing first-derivative matrices that are $O(\Delta x^2)$ accurate. In this case, Table 1 applies. The simplest case to consider is when *Dirichlet boundary conditions* are applied, i.e. $v(a) = v(b) = 0$. The Dirichlet boundaries imply that only the interior points $v(x_j)$, where $j = 1, 2, \cdots, N$ must be solved for. The resulting derivative matrix is given by

$$\frac{\partial}{\partial x} \text{ with } u(a)=u(b)=0 \;\rightarrow\; \mathbf{A}_1 = \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 0 & \cdots & & 0 & -1 & 0 \end{bmatrix}. \tag{8.8.9}$$

If the boundary conditions are changed to *Neumann boundary conditions* with $dv(a)/dx = dv(b)/dx = 0$, then the derivative matrix changes. Specifically, if use is made of the forward- and backward-difference formulas of Table 3.3, then it is easy to show that

$$\frac{dv_0}{dx} = \frac{-3v_0+4v_1-v_2}{2\Delta x} = 0 \;\rightarrow\; v_0 = \frac{4}{3}v_1 - \frac{1}{3}v_2. \tag{8.8.10}$$

Similarly at the right boundary, one can find

$$\frac{dv_{N+1}}{dx} = \frac{-3v_{N+1}+4v_N-v_{N-1}}{2\Delta x} = 0 \;\rightarrow\; v_{N+1} = \frac{4}{3}v_N - \frac{1}{3}v_{N-1}. \tag{8.8.11}$$

Thus, both boundary values, v_0 and v_{N+1}, are determined from the interior points alone. The final piece of information necessary is the following:

$$\frac{dv_1}{dx} = \frac{v_2 - v_0}{2\Delta x} = \frac{v_2 - [(4/3)v_1 - (1/3)v_2]}{2\Delta x} = \frac{(4/3)(v_2 - v_1)}{2\Delta x}. \tag{8.8.12}$$

A similar calculation can be done for dv_N/dx to yield the derivative matrix

$$\frac{\partial}{\partial x} \text{ with } \frac{du(a)}{dx} = \frac{du(b)}{dx} = 0 \;\rightarrow\; \mathbf{A}_2 = \frac{1}{2\Delta x}\begin{bmatrix} -4/3 & 4/3 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 0 & \cdots & & 0 & -4/3 & 4/3 \end{bmatrix}. \tag{8.8.13}$$

Finally, we consider the case of *periodic boundary conditions* for which $v(a) = v(b)$. This implies that $v_0 = v_{N+1}$. Thus, when considering this case, the unknowns include v_0 through v_N, i.e. there are $N+1$ unknowns as opposed to N unknowns. The derivative matrix in this case is

$$\frac{\partial}{\partial x} \text{ with } u(a)=u(b) \;\rightarrow\; \mathbf{A}_3 = \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & -1 \\ -1 & 0 & 1 & 0 & \cdots & \\ 0 & -1 & 0 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & -1 & 0 & 1 \\ 1 & \cdots & & 0 & -1 & 0 \end{bmatrix} \tag{8.8.14}$$

where the top right and bottom left corners come from the fact that, for instance, $dv_N/dx = (v_{N+1} - v_{N-1})/(2\Delta x) = (v_0 - v_{N-1})/(2\Delta x)$.

Second-derivative analogues can also be created for these matrices using the same basic ideas as for first derivatives. In this case, we find for Dirichlet boundary conditions

$$\frac{\partial^2}{\partial x^2} \text{ with } u(a)=u(b)=0 \;\rightarrow\; \mathbf{B}_1 = \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 0 & \cdots & & 0 & 1 & -2 \end{bmatrix}. \tag{8.8.15}$$

For Neumann boundary conditions, the following applies:

$$\frac{\partial^2}{\partial x^2} \text{ with } \frac{du(a)}{dx} = \frac{du(b)}{dx} = 0 \;\rightarrow\; \mathbf{B}_2 = \frac{1}{\Delta x^2}\begin{bmatrix} -2/3 & 2/3 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 0 & \cdots & & 0 & 2/3 & -2/3 \end{bmatrix} \tag{8.8.16}$$

where the boundaries are again evaluated from the interior points through the relations (8.8.10) and (8.8.11). Finally, for periodic boundaries, the matrix is

$$\frac{\partial^2}{\partial x^2} \text{ with } u(a)=u(b) \;\rightarrow\; \mathbf{B}_3 = \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 1 \\ 1 & -2 & 1 & 0 & \cdots & \\ 0 & 1 & -2 & 1 & \cdots & \\ & & \vdots & & & \\ & \cdots & & 1 & -2 & 1 \\ 1 & \cdots & & 0 & 1 & -2 \end{bmatrix}. \tag{8.8.17}$$

The derivative operators shown here are fairly standard and can easily be implemented in Python. The following code constructs matrices $\mathbf{A}_1$–$\mathbf{A}_3$ and $\mathbf{B}_1$–$\mathbf{B}_3$. First, there is the construction of the first-derivative matrices:

```
# Assume N and dx given
A = np.zeros((N, N))
for j in range(N - 1):
    A[j, j + 1] = 1
    A[j + 1, j] = -1
A1 = A / (2 * dx) # Dirichlet matrices

A2 = np.copy(A)
A2[0, 0] = -4 / 3
A2[0, 1] = 4 / 3
A2[N - 1, N - 1] = 4 / 3
A2[N - 1, N - 2] = -4 / 3
A2 = A2 / (2 * dx) # Neumann matrices

A3 = np.copy(A)
A3[N - 1, 0] = 1
A3[0, N - 1] = -1
A3 = A3 / (2 * dx) # Periodic BC matrices
```

The second-derivative matrices can be constructed similarly with the following Python code:

```
B = np.zeros((N, N))
for j in range(N):
    B[j, j] = -2
for j in range(N - 1):
    B[j, j + 1] = 1
    B[j + 1, j] = 1
B1 = B / (dx**2) # Dirichlet matrices for B

B2 = np.copy(B)
B2[0, 0] = -2 / 3
B2[0, 1] = 2 / 3
B2[N - 1, N - 1] = -2 / 3
B2[N - 1, N - 2] = 2 / 3
B2 = B2 / (dx**2) # Neumann matrices for B

B3 = np.copy(B)
B3[N - 1, 0] = 1
B3[0, N - 1] = -1
B3 = B3 / (dx**2) # Periodic BC matrices for B
```

Given how often such derivative matrices are used, the above code could be used as part of function call to simply pull out the derivative matrix required. In addition, we will later learn how to encode such matrices in a sparse manner since they are mostly zeros.

8.8.3 Example: The harmonic oscillator

To see an example of how to compute the spectra of a linear operator, consider the example of the harmonic oscillator which has the well-known Hermite function solutions. This is a Sturm–Liouville problem with $p(x) = w(x) = 1$ and $q(x) = x^2$. The $q(x)$ plays the role of a trapping potential in quantum mechanics. Typically, the harmonic oscillator is specified on the domain $x \in [-\infty, \infty]$ with solutions $v(\pm\infty) \to 0$. Instead of specifying that the domain is infinite, we will consider a finite domain of size $x \in [-a, a]$ with the Dirichlet boundary conditions $v(\pm L) = 0$. Thus, we have the eigenvalue problem

$$\mathcal{L}v = -\frac{d^2v}{dx^2} + x^2 v = \lambda v \quad \text{with} \quad v(\pm L) = 0. \tag{8.8.18}$$

Our objective is to construct the spectra and eigenfunctions of this linear operator. We can even compare them directly to the known solutions to see how accurate the reconstruction is. The following code discretizes the domain and computes the spectra.

```
L = 4  # domain size
N = 200  # discretization of interior
x = np.linspace(-L, L, N + 2)  # add boundary points
dx = x[1] - x[0]  # compute dx

P = np.zeros((N, N)) # Compute P matrix
for j in range(N):
    P[j, j] = x[j + 1] ** 2  # potential x^2


linL = -B1 + P # Compute linear operator


D,V = eig(linL) # Compute eigenvalues/eigenvectors

sorted_indices = np.argsort(np.abs(D))[::-1]
Dsort = D[sorted_indices]
Vsort = V[:, sorted_indices]

D5 = Dsort[N-5:N]
V5 = Vsort[:,N-5:N]
```

This code computes the second-order accurate spectra of the harmonic oscillator. Here, $\Delta x \approx 0.04$ so that an accuracy of 10^{-3} is expected. Only the five smallest eigenvalues

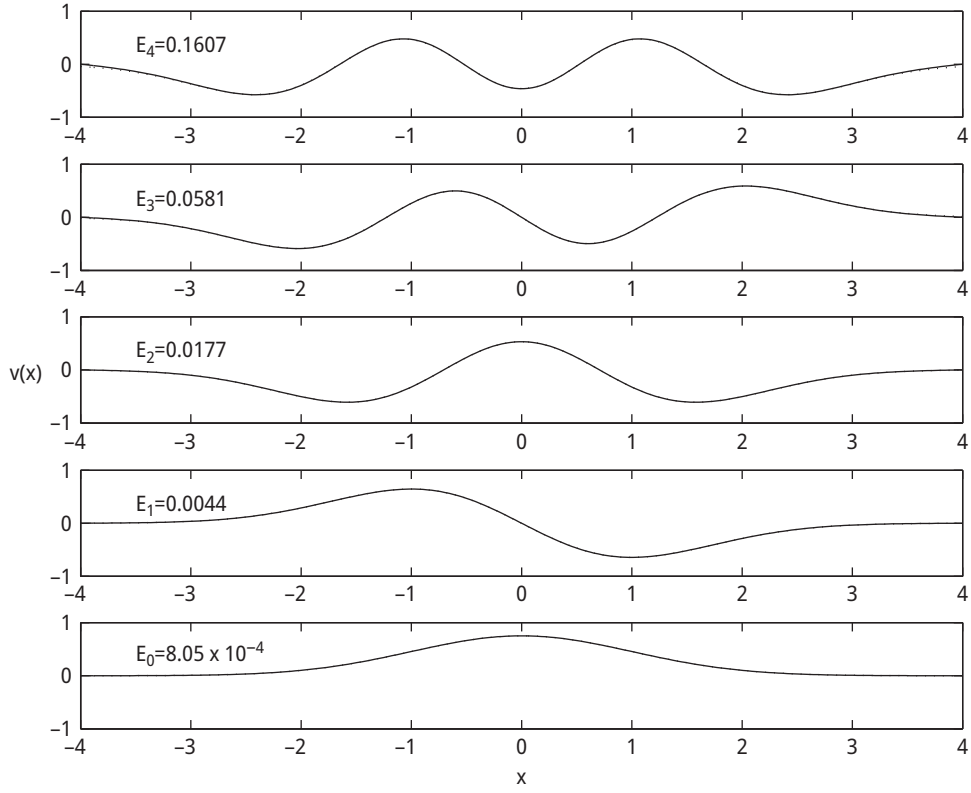


Figure 8.14 First five eigenfunctions of the harmonic oscillator computed using finite difference discretization. Shown are the computed eigenfunctions (solid line) versus the exact solution (dotted line). The two lines are almost indistinguishable from each other. Thus, the norm of the difference of the solutions is given by E_j for the different eigenfunctions.

are returned in the code. These five eigenvalues are theoretically known to be given by $\lambda_n = (2n+1)$ with $n = 0, 1, 2, \ldots$. In our calculation, we find the first five $\lambda_n = 1, 3, 5, 7, 9 \approx 0.9999, 2.9995, 4.9991, 7.0009, 9.0152$. Thus the accuracy is as prescribed. The first five eigenfunctions are shown in Fig. 8.14. As can be seen, the finite difference method is fairly easy to implement and yields fast and accurate results.

8.9 Neural Networks for Time-Stepping

We return to the basic time-stepping algorithms introduced at the start of this chapter. The algorithms developed previously, including simple Euler stepping and Runge–Kutta, all involve approximating a Δt step into the future using Taylor series expansions. An

alternative way to think about a time-stepper is as a flow map from time t to time $t + \Delta t$ [34; 35]. Neural networks can learn such a flow map. Specifically, we can consider the basic input–output relationship of a neural network to be a flow map from time t to time $t + \Delta$:

$$\mathbf{Y} = f_{\boldsymbol{\theta}}(\mathbf{X}) \quad \rightarrow \quad \mathbf{y}_{n+1} = f_{\boldsymbol{\theta}}(\mathbf{y}_n) \tag{8.8.19}$$

where $f_{\boldsymbol{\theta}}(\cdot)$ is a trained neural network. Thus, given trajectories of (8.1.1), the neural network weights $\boldsymbol{\theta}$ are updated to minimize the time-stepping error given a specific neural network architecture. Such an architecture effectively learns the right-hand side function of (8.1.1) [36–39].

As with all neural networks, training data is required in order to determine the weights of the activation functions. Figure 8.15 shows an example feed forward architecture mapping the input $\mathbf{y}_n$ to $p\Delta t$ steps into the future $\tilde{\mathbf{y}}_{n+p}$ by minimizing the error $\|\mathbf{y}_{n+p} - \tilde{\mathbf{y}}_{n+p}\|$. In what follows, we train a neural network to take one step forward so that $p = 1$. The training data comprises 100 trajectories with randomly generated initial conditions. The data is mapped to input and output data which correspond to $\mathbf{y}_n$ and $\mathbf{y}_{n+1}$, respectively. The input–output data matrices are then made into torch objects.

```
def lorenz(t, x, sig=10, r=28, b=8/3):
    return [sig*(x[1]-x[0]), r*x[0]-x[1]-x[0]*x[2],
        x[0]*x[1]-b*x[2]]

dt = 0.01; T = 8; t = np.arange(0, T + dt, dt); n_trajectories = 100

input_data = []; output_data = []

for j in range(n_trajectories):
    x0 = 30 * (np.random.rand(3) - 0.5)
    sol = solve_ivp(lorenz, [0, T], x0, t_eval=t, atol=1e-11,
        rtol=1e-10)
    y = sol.y.T
    input_data.append(y[:-1])
    output_data.append(y[1:])

input_data = np.vstack(input_data)
output_data = np.vstack(output_data)

input_tensor = torch.tensor(input_data, dtype=torch.float32)
output_tensor = torch.tensor(output_data, dtype=torch.float32)
```

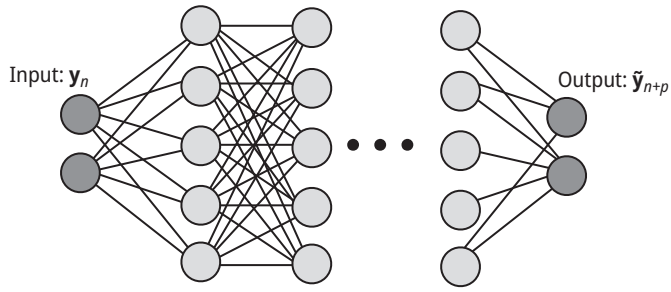


Figure 8.15 Feed forward neural network used as an approximation for time-stepping. The input $\mathbf{y}_n$ is mapped through a network to p steps into the future $\tilde{\mathbf{y}}_{n+p}$ by minimizing the error $\|\mathbf{y}_{n+p} - \tilde{\mathbf{y}}_{n+p}\|$.

With the training data generated, a feed forward neural network is constructed. In this specific case, it is a three-hidden-layer network that takes the original three-dimensional input space to 10 dimensions. The last layer is a linear layer that brings the output back to three dimensions. The activation functions are varied with both a sigmoid and ReLu layer being used, for instance. The following code defines the neural network class and structure.

```
class FeedforwardNN(nn.Module):
    def _ _init_ _(self):
        super(FeedforwardNN, self)._ _init_ _()
        self.fc1 = nn.Linear(3, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 10)
        self.fc4 = nn.Linear(10, 3)
        self.sigmoid = nn.Sigmoid()
        self.radbas = nn.ReLU()
        self.linear = nn.Identity()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.radbas(self.fc2(x))
        x = self.linear(self.fc3(x))
        x = self.fc4(x)
        return x
```

Training now requires a loss function and various hyperparameter settings, which include the number of epochs, batch size, loss rate, optimizer, and error function. These can all be easily adjusted in the training.

```
net = FeedforwardNN()
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
n_epochs = 200; batch_size = 32

for epoch in range(n_epochs):
    permutation = torch.randperm(input_tensor.size()[0])
    for i in range(0, input_tensor.size()[0], batch_size):
        indices = permutation[i:i + batch_size]
        batch_input, batch_output = input_tensor[indices],
        output_tensor[indices]

        optimizer.zero_grad()
        outputs = net(batch_input)
        loss = criterion(outputs, batch_output)
        loss.backward()
        optimizer.step()
```

Given the number of hyperparameters, it is often the case that hyperparameter sweeps are performed in order to increase the performance of the neural network. Although this particular neural network is small and can be easily trained in minutes, hyperparameter tuning on larger problems takes substantial amount of time, effort, and patience.

Once the neural network is trained, it can be deployed with new initial data as shown in Fig. 8.16. The following code uses the neural network to *unroll* to future states. These can then be compared to actual simulations to verify their behaviour.

```
ynn = [x0]
x0_tensor = torch.tensor(x0, dtype=torch.float32).unsqueeze(0)
for _ in range(1, len(t)):
    y0_tensor = net(x0_tensor)
    ynn.append(y0_tensor.detach().numpy().flatten())
    x0_tensor = y0_tensor

ynn = np.array(ynn)
```

The trained neural network can be deployed as a proxy for a numerical time-stepping algorithm such as Runge–Kutta. Unlike standard numerical steppers, however, neural networks rarely have rigorous estimates for convergence. Rather, cross-validation is used for

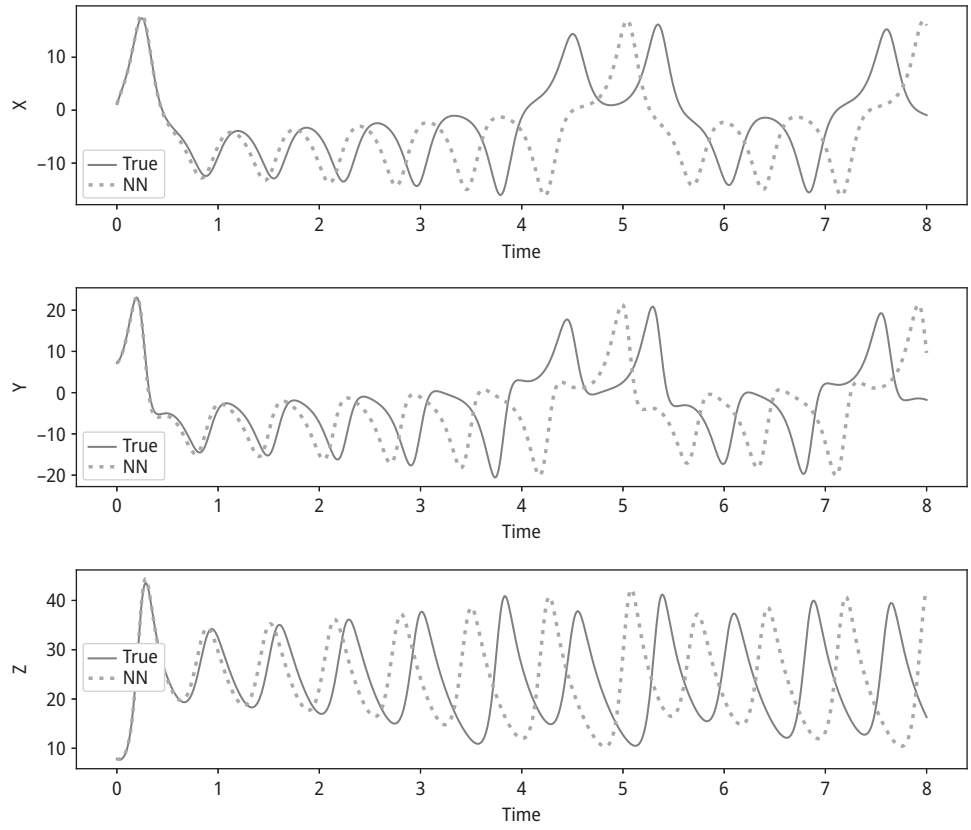


Figure 8.16 Learned neural network for time-stepping. A new (test) initial condition is used to roll out the trajectory to the future. The salient features are learned. However, it is expected for Lorenz equation that the solution would diverge from the true trajectory due to sensitivity to initial conditions. More training data, deeper and wider networks, and often longer training times can greatly improve the neural networks' capabilities.

determining the convergence properties. More sophisticated algorithms can also be applied to learn multiscale features for time-stepping [34]. Additionally, such techniques can be used in equation-free architectures, where no governing equations are known but data is generated from sensors directly. This is something that is not possible with numerical steppers, which require governing equations.

Finite Difference Methods

9

9.1 Finite Difference Discretization 219
9.2 Advanced Iterative Solution Methods for $\mathbf{Ax} = \mathbf{b}$ 225
9.3 Fast Poisson Solvers: The Fourier Transform 225
9.3.1 The fast Fourier transform 227
9.3.2 The streamfunction 227
9.3.3 Python commands 228
9.4 Comparison of Solution Techniques for $\mathbf{Ax} = \mathbf{b}$: Rules of Thumb 229
9.4.1 Fast Fourier transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT 229
9.4.2 Sparse matrices: SPDIAGS, SPY 232
9.4.3 Iterative methods: BICGSTAB, GMRES 234
9.5 Overcoming Computational Difficulties 235
9.5.1 Streamfunction equations: Nonuniqueness 235
9.5.2 Fast Fourier transforms: Divide by zero 236
9.5.3 Sparse derivative matrices: Advection terms 237

Finite difference methods are based exclusively on Taylor expansions. They are one of the most powerful methods available since they are relatively easy to implement, can handle fairly complicated boundary conditions, and allow for explicit calculations of the computational error. The result of discretizing any given problem is the need to solve a large linear system of equations or perhaps manipulate large, sparse matrices. All this will be dealt with in the following sections.

9.1 Finite Difference Discretization

To discuss the solution of a given problem with the finite difference method, we consider a specific example from atmospheric sciences, which is detailed in the exercises at the end of this chapter. The quasi-two-dimensional motion of the atmosphere can be modelled by the advection–diffusion behaviour for the vorticity $\omega(x, y, t)$, which is coupled to the streamfunction $\psi(x, y, t)$:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{9.1.1a}$$

$$\nabla^2 \psi = \omega \tag{9.1.1b}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0010

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{9.1.2}$$

and $\nabla^2 = \partial_x^2 + \partial_y^2$ is the two-dimensional Laplacian. Note that this equation has both an advection component (hyperbolic) from $[\psi, \omega]$ and a diffusion component (parabolic) from $\nu\nabla^2\omega$. We will assume that we are given the initial value of the vorticity

$$\omega(x, y, t = 0) = \omega_0(x, y)\,. \tag{9.1.3}$$

Additionally, we will proceed to solve this problem with periodic boundary conditions. This gives the following set of boundary conditions

$$\omega(-L, y, t) = \omega(L, y, t) \tag{9.1.4a}$$

$$\omega(x, -L, t) = \omega(x, L, t) \tag{9.1.4b}$$

$$\psi(-L, y, t) = \psi(L, y, t) \tag{9.1.4c}$$

$$\psi(x, -L, t) = \psi(x, L, t) \tag{9.1.4d}$$

where we are solving on the computational domain $x \in [-L, L]$ and $y \in [-L, L]$.

Basic algorithm structure: Before discretizing the governing partial differential equation, it is important to clarify what the basic solution procedure will be. Two physical quantities need to be solved as functions of time:

$$\psi(x, y, t) \quad \text{streamfunction} \tag{9.1.5a}$$

$$\omega(x, y, t) \quad \text{vorticity.} \tag{9.1.5b}$$

We are given the initial vorticity $\omega_0(x, y)$ and periodic boundary conditions. The solution procedure is as follows:

(1) **Elliptic solve**: Solve the elliptic problem $\nabla^2\psi = \omega_0$ to find the streamfunction at time zero $\psi(x, y, t = 0) = \psi_0$.

(2) **Time-stepping**: Given initial ω_0 and ψ_0, solve the advection–diffusion problem by time-stepping with a given method. The Euler method is illustrated below

$$\omega(x, y, t + \Delta t) = \omega(x, y, t) + \Delta t \left(\nu\nabla^2\omega(x, y, t) - [\psi(x, y, t), \omega(x, y, t)]\right).$$

This advances the solution Δt into the future.

(3) **Loop**: With the updated value of $\omega(x, y, \Delta t)$, we can repeat the process by again solving for $\psi(x, y, \Delta t)$ and updating the vorticity once again.

This gives the basic algorithmic structure, which must be implemented in order to generate the solution for the vorticity and streamfunction as functions of time. It only remains to discretize the problem and solve.

Step 1: Elliptic solve: We begin by discretizing the elliptic solve problem for the streamfunction $\psi(x, y, t)$. The governing equation in this case is

$$\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = \omega. \tag{9.1.6}$$

Using the central difference formulas of Section 3.1 reduces the governing equation to a set of linearly coupled equations. In particular, we find for a second-order accurate central difference scheme that the elliptic equation reduces to

$$\frac{\psi(x+\Delta x,y,t) - 2\psi(x,y,t) + \psi(x-\Delta x,y,t)}{\Delta x^2} + \frac{\psi(x,y+\Delta y,t) - 2\psi(x,y,t) + \psi(x,y-\Delta y,t)}{\Delta y^2} = \omega(x,y,t). \tag{9.1.7}$$

Thus, the solution at each point depends upon itself and four neighbouring points. This creates a five-point stencil for solving this equation. Figure 9.1 illustrates the stencil that arises from discretization. For convenience, we denote

$$\psi_{mn} = \psi(x_m, y_n, t)\,. \tag{9.1.8}$$

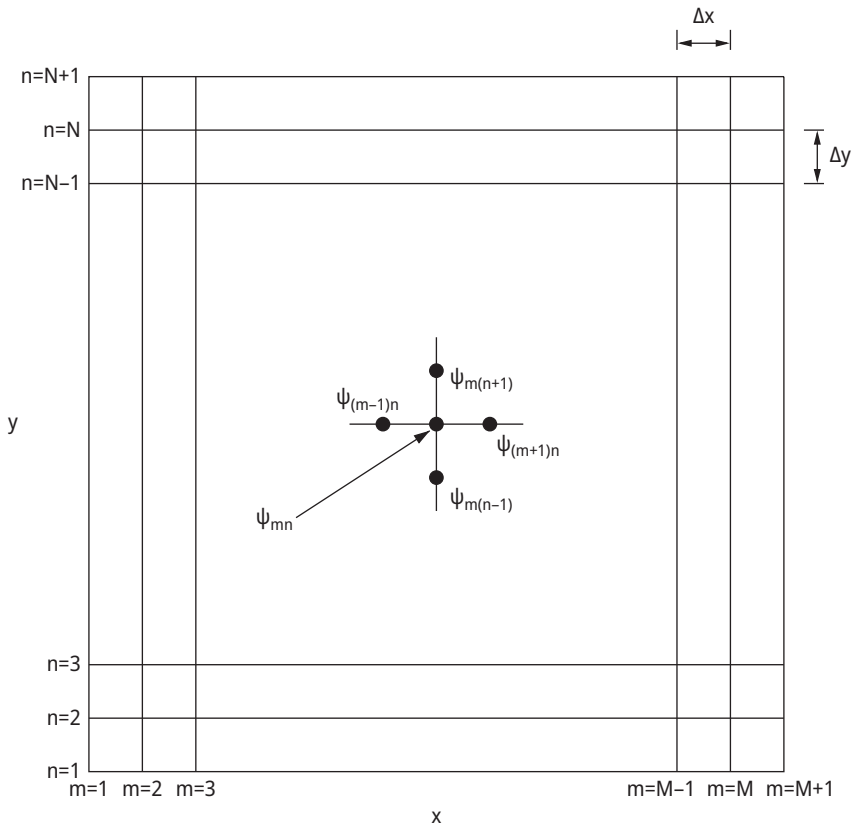


Figure 9.1 Discretization stencil for solving for the streamfunction with second-order accurate central difference schemes. Note that $\psi_{mn} = \psi(x_m, y_n)$.

By letting $\Delta x^2 = \Delta y^2 = \delta^2$, the discretized equations reduce to

$$-4\psi_{mn} + \psi_{(m-1)n} + \psi_{(m+1)n} + \psi_{m(n-1)} + \psi_{m(n+1)} = \delta^2 \omega_{mn} \tag{9.1.9}$$

with periodic boundary conditions imposing the following constraints

$$\psi_{1n} = \psi_{(N+1)n} \tag{9.1.10a}$$

$$\psi_{m1} = \psi_{m(N+1)} \tag{9.1.10b}$$

where $N+1$ is the total number of discretization points in the computational domain in both the x- and y-directions.

As a simple example, consider the four-point system for which $N = 4$. For this case, we have the following sets of equations:

$$\begin{aligned}
-4\psi_{11} + \psi_{41} + \psi_{21} + \psi_{14} + \psi_{12} &= \delta^2 \omega_{11} \\
-4\psi_{12} + \psi_{42} + \psi_{22} + \psi_{11} + \psi_{13} &= \delta^2 \omega_{12} \\
&\vdots \\
-4\psi_{21} + \psi_{11} + \psi_{31} + \psi_{24} + \psi_{22} &= \delta^2 \omega_{21} \\
&\vdots
\end{aligned} \tag{9.1.11}$$

which results in the sparse matrix (banded matrix) system

$$\mathbf{A}\psi = \delta^2 \omega \tag{9.1.12}$$

where

$$\mathbf{A} = \begin{bmatrix}
-4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4
\end{bmatrix} \tag{9.1.13}$$

and

$$\psi = (\psi_{11}\,\psi_{12}\,\psi_{13}\,\psi_{14}\,\psi_{21}\,\psi_{22}\,\psi_{23}\,\psi_{24}\,\psi_{31}\,\psi_{32}\,\psi_{33}\,\psi_{34}\,\psi_{41}\,\psi_{42}\,\psi_{43}\,\psi_{44})^{\mathrm{T}} \tag{9.1.14a}$$

$$\omega = (\omega_{11}\,\omega_{12}\,\omega_{13}\,\omega_{14}\,\omega_{21}\,\omega_{22}\,\omega_{23}\,\omega_{24}\,\omega_{31}\,\omega_{32}\,\omega_{33}\,\omega_{34}\,\omega_{41}\,\omega_{42}\,\omega_{43}\,\omega_{44})^{\mathrm{T}}\,. \tag{9.1.14b}$$

Any matrix solver can then be used to generate the values of the two-dimensional streamfunction, which are contained completely in the vector ψ.

Step 2: Time-stepping: After generating the matrix $\mathbf{A}$ and the value of the streamfunction $\psi(x,y,t)$, we use this updated value along with the current value of the vorticity to take a time-step Δt into the future. The appropriate equation is the advection–diffusion evolution equation:

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega\,. \tag{9.1.15}$$

Using the definition of the bracketed term and the Laplacian, this equation is

$$\frac{\partial \omega}{\partial t} = \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} + \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right)\,. \tag{9.1.16}$$

Second-order central-differencing discretization then yields

$$\begin{aligned}\frac{\partial \omega}{\partial t} = &\left(\frac{\psi(x,y+\Delta y,t)-\psi(x,y-\Delta y,t)}{2\Delta y}\right)\left(\frac{\omega(x+\Delta x,y,t)-\omega(x-\Delta x,y,t)}{2\Delta x}\right)\\ &-\left(\frac{\psi(x+\Delta x,y,t)-\psi(x-\Delta x,y,t)}{2\Delta x}\right)\left(\frac{\omega(x,y+\Delta y,t)-\omega(x,y-\Delta y,t)}{2\Delta y}\right)\\ &+\nu\left\{\frac{\omega(x+\Delta x,y,t)-2\omega(x,y,t)+\omega(x-\Delta x,y,t)}{\Delta x^2}\right.\\ &\left.+\frac{\omega(x,y+\Delta y,t)-2\omega(x,y,t)+\omega(x,y-\Delta y,t)}{\Delta y^2}\right\}.\end{aligned} \tag{9.1.17}$$

This is simply a large system of differential equations, which can be stepped forward in time with any convenient time-stepping algorithm such as fourth-order Runge–Kutta. In particular, given that there are $N+1$ points and periodic boundary conditions, this reduces the system of differential equations to an $N \times N$ coupled system. Once we have updated the value of the vorticity, we must again update the value of the streamfunction to once again update the vorticity. This loop continues until the solution at the desired future time is achieved. Figure 9.2 illustrates how the five-point, two-dimensional stencil advances the solution.

The behaviour of the vorticity is illustrated in Fig. 9.3, where the solution is advanced for eight time units. The initial condition used in this simulation is

$$\omega_0 = \omega(x,y,t=0) = \exp\left(-2x^2 - \frac{y^2}{20}\right)\,. \tag{9.1.18}$$

This stretched Gaussian is seen to rotate while advecting and diffusing vorticity. Multiple vortex solutions can also be considered along with oppositely signed vortices.

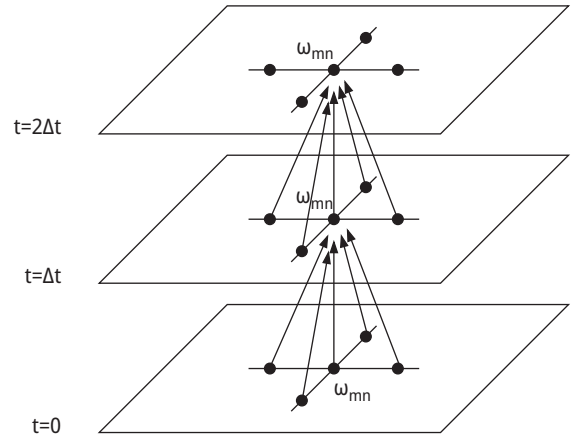


Figure 9.2 Discretization stencil resulting from centre-differencing of the advection–diffusion equations. Note that for explicit stepping schemes the future solution only depends upon the present. Thus, we are not required to solve a large linear system of equations.

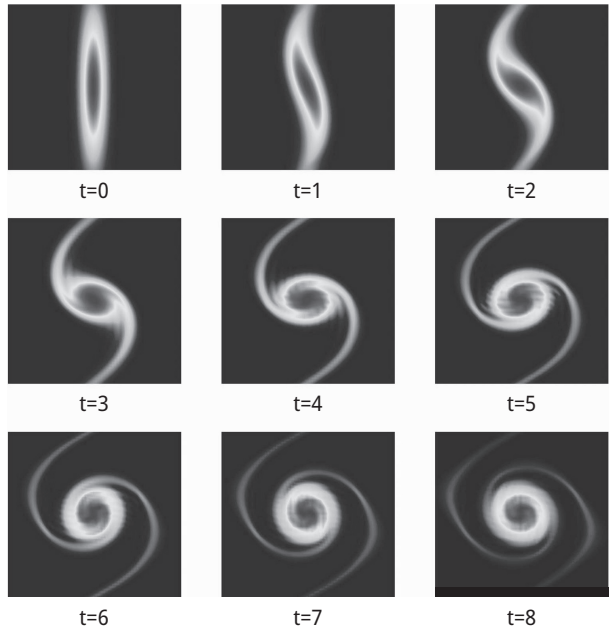


Figure 9.3 Time evolution of the vorticity $\omega(x, y, t)$ over eight time units with $\nu = 0.001$ and a spatial domain $x \in [-10, 10]$ and $y \in [-10, 10]$. The initial condition was a stretched Gaussian of the form $\omega(x, y, 0) = \exp(-2x^2 - y^2/20)$.

9.2 Advanced Iterative Solution Methods for $\mathbf{Ax} = \mathbf{b}$

In addition to the standard techniques of Gaussian elimination or LU decomposition for solving $\mathbf{Ax} = \mathbf{b}$, a wide range of iterative techniques are available.

Application to advection–diffusion: When discretizing many systems of interest, such as the advection–diffusion problem, we are left with a system of equations that is naturally geared towards iterative methods. Discretization of the stream/function previously yielded the system

$$-4\psi_{mn} + \psi_{(m+1)n} + \psi_{(m-1)n} + \psi_{m(n+1)} + \psi_{m(n-1)} = \delta^2\omega_{mn}\,. \tag{9.2.1}$$

The matrix $\mathbf{A}$ in this case is represented by the left-hand side of the equation. Letting ψ_{mn} be the diagonal term, the iteration procedure yields

$$\psi_{mn}^{k+1} = \frac{\psi_{(m+1)n}^{k} + \psi_{(m-1)n}^{k} + \psi_{m(n+1)}^{k} + \psi_{m(n-1)}^{k} - \delta^2\omega_{mn}}{4}\,. \tag{9.2.2}$$

Note that the diagonal term has a coefficient of $|-4| = 4$, and the sum of the off-diagonal elements is $|1| + |1| + |1| + |1| = 4$. Thus, the system is at the borderline of being diagonally dominant. So although convergence is not guaranteed, it is highly likely that we could get the Jacobi scheme to converge.

Finally, we consider the operation count associated with the iteration methods. This will allow us to compare this solution technique with Gaussian elimination and LU decomposition. The following basic algorithmic steps are involved:

(1) Update each ψ_{mn} which costs N operations times the number of nonzero diagonals D.
(2) For each ψ_{mn}, perform the appropriate additions and subtractions. In this case, there are five operations.
(3) Iterate until the desired convergence which costs K operations.

Thus, the total number of operations is $O(N \cdot D \cdot 5 \cdot K)$. If the number of iterations K can be kept small, then iteration provides a viable alternative to the direct solution techniques.

9.3 Fast Poisson Solvers: The Fourier Transform

Other techniques exist for solving many computational problems, which are not based upon the standard Taylor series discretization. For instance, we have considered solving the streamfunction equation

$$\nabla^2\psi = \omega \tag{9.3.1}$$

by discretizing in both the x- and y-directions and solving the associated linear problem $\mathbf{Ax} = \mathbf{b}$. At best, we can use a factorization scheme to solve this problem in $O(N^2)$ operations. Although iteration schemes have the possibility of outperforming this, it is not guaranteed.

Another alternative is to use the fast Fourier transform (FFT). The FFT is an integral transform defined over the entire line $x \in [-\infty, \infty]$. Given computational practicalities, however, we transform over a finite domain $x \in [-L, L]$ and assume periodic boundary conditions due to the oscillatory behaviour of the kernel of the Fourier transform. The Fourier transform and its inverse can be defined as

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{9.3.2a}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} F(k) dk \,. \tag{9.3.2b}$$

There are other equivalent definitions. However, this definition will serve to illustrate the power and functionality of the Fourier transform method. We again note that formally, the transform is over the entire real line $x \in [-\infty, \infty]$, whereas our computational domain is only over a finite domain $x \in [-L, L]$. Further, the kernel of the transform, $\exp(\pm ikx)$, describes oscillatory behaviour. Thus, the Fourier transform is essentially an eigenfunction expansion over all continuous wavenumbers k. And once we are on a finite domain $x \in [-L, L]$, the continuous eigenfunction expansion becomes a discrete sum of eigenfunctions and associated wavenumbers (eigenvalues).

Derivative relations: The critical property in the usage of Fourier transforms concerns derivative relations. To see how these properties are generated, we begin by considering the Fourier transform of $f'(x)$. We denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$. Thus, we find

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f'(x) dx = f(x) e^{-ikx}|_{-\infty}^{\infty} + \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \,. \tag{9.3.3}$$

Assuming that $f(x) \to 0$ as $x \to \pm\infty$ results in

$$\widehat{f'(x)} = \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx = ik\widehat{f(x)} \,. \tag{9.3.4}$$

Thus, the basic relation $\widehat{f'} = ik\widehat{f}$ is established. It is easy to generalize this argument to an arbitrary number of derivatives. The final result is the following relation between Fourier transforms of the derivative and the Fourier transform itself

$$\widehat{f^{(n)}} = (ik)^n \widehat{f} \,. \tag{9.3.5}$$

This property is what makes Fourier transforms so useful and practical.

As an example of the Fourier transform, consider the following differential equation

$$y'' - \omega^2 y = -f(x) \quad x \in [-\infty, \infty] \,. \tag{9.3.6}$$

We can solve this by applying the Fourier transform to both sides. This gives the following reduction

$$\begin{aligned}
\widehat{y''} - \omega^2 \widehat{y} &= -\widehat{f} \\
-k^2 \widehat{y} - \omega^2 \widehat{y} &= -\widehat{f} \\
(k^2 + \omega^2)\widehat{y} &= \widehat{f} \\
\widehat{y} &= \frac{\widehat{f}}{k^2 + \omega^2}.
\end{aligned} \tag{9.3.7}$$

To find the solution $y(x)$, we invert the last expression above to yield

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} \frac{\widehat{f}}{k^2 + \omega^2} dk. \tag{9.3.8}$$

This gives the solution in terms of an integral which can be evaluated analytically or numerically.

9.3.1 The fast Fourier transform

The fast Fourier transform routine was developed specifically to perform the forward and backward Fourier transforms. In the mid-1960s, Cooley and Tukey developed what is now commonly known as the FFT algorithm [40]. Their algorithm was named one of the top 10 algorithms of the twentieth century for one reason: the operation count for solving a system dropped to $O(N \log N)$. For N large, this operation count grows almost linearly like N. Thus, it represents a great leap forward from Gaussian elimination and LU decomposition. The key features of the FFT routine are as follows:

(1) It has a low operation count: $O(N \log N)$.

(2) It finds the transform on an interval $x \in [-L, L]$. Since the integration kernel $\exp(\pm ikx)$ is oscillatory, it implies that the solutions on this finite interval have periodic boundary conditions.

(3) The key to lowering the operation count to $O(N \log N)$ is in discretizing the range $x \in [-L, L]$ into 2^n points, i.e. the number of points should be $2, 4, 8, 16, 32, 64, 128, 256, \cdots$.

(4) The FFT has excellent accuracy properties, typically well beyond that of standard discretization schemes.

We will consider the underlying FFT algorithm in detail at a later time. For more information at the present, see [40] for a broader overview.

9.3.2 The streamfunction

The FFT algorithm provides a fast and efficient method for solving the streamfunction equation

$$\nabla^2 \psi = \omega \tag{9.3.9}$$

given the vorticity ω. In two dimensions, this equation is equivalent to

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \omega \,. \tag{9.3.10}$$

Denoting the Fourier transform in x as $\widehat{f(x)}$ and the Fourier transform in y as $\widetilde{g(y)}$, we transform the equation. We begin by transforming in x:

$$\widehat{\frac{\partial^2 \psi}{\partial x^2}} + \widehat{\frac{\partial^2 \psi}{\partial y^2}} = \widehat{\omega} \quad \rightarrow \quad -k_x^2 \widehat{\psi} + \frac{\partial^2 \widehat{\psi}}{\partial y^2} = \widehat{\omega} \,, \tag{9.3.11}$$

where k_x are the wavenumbers in the x-direction. Transforming now in the y-direction gives

$$-k_x^2 \widetilde{\widehat{\psi}} + \widetilde{\frac{\partial^2 \widehat{\psi}}{\partial y^2}} = \widetilde{\widehat{\omega}} \quad \rightarrow \quad -k_x^2 \widetilde{\widehat{\psi}} + -k_y^2 \widetilde{\widehat{\psi}} = \widetilde{\widehat{\omega}} \,. \tag{9.3.12}$$

This can be rearranged to obtain the final result

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2} \,. \tag{9.3.13}$$

The remaining step is to inverse-transform in x and y to get back to the solution $\psi(x, y)$.

There is one mathematical difficulty that must be addressed. The streamfunction equation with periodic boundary conditions does not have a unique solution. Thus, if $\psi_0(x, y, t)$ is a solution, so is $\psi_0(x, y, t) + c$, where c is an arbitrary constant. When solving this problem with FFTs, the FFT will arbitrarily add a constant to the solution. Fundamentally, we are only interested in derivatives of the streamfunction. Therefore, this constant is inconsequential. When solving with direct methods for $\mathbf{Ax} = \mathbf{b}$, the nonuniqueness gives a singular matrix $\mathbf{A}$. Thus, solving with Gaussian elimination, LU decomposition or iterative methods is problematic. But since the arbitrary constant does not matter, we can simply pin the streamfunction to some prescribed value on our computational domain. This will fix the constant c and give a unique solution to $\mathbf{Ax} = \mathbf{b}$. For instance, we could impose the following constraint condition $\psi(-L, -L, t) = 0$. Such a condition pins the value of $\psi(x, y, t)$ at the left-hand corner of the computational domain and fixes c.

9.3.3 Python commands

The commands for executing the fast Fourier transform and its inverse are as follows:

- $fft(x)$: Forward Fourier transform a vector $\mathbf{x}$.
- $ifft(x)$: Inverse Fourier transform a vector $\mathbf{x}$.

9.4 Comparison of Solution Techniques for $\mathbf{Ax} = \mathbf{b}$: Rules of Thumb

The practical implementation of the mathematical tools available in Python is crucial. This section will focus on the use of some of the more sophisticated routines in Python, which are cornerstones to scientific computing. Included in this section will be a discussion of the fast Fourier transform routines (*fft, ifft, fftshift, ifftshift, fft2, ifft2*), sparse matrix construction (*spdiag, spy*), and high-end iterative techniques for solving $\mathbf{Ax} = \mathbf{b}$ (*bicgstab, gmres*). These routines should be studied carefully since they are the building blocks of any serious scientific computing code.

9.4.1 Fast Fourier transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT

The fast Fourier transform will be the first subject discussed. Its implementation is straightforward. Given a function, which has been discretized with 2^n points and represented by a vector $\mathbf{x}$, the FFT is found with the command *fft(x)*. Aside from transforming the function, the algorithm associated with the FFT does three major things: it shifts the data so that $x \in [0, L] \to [-L, 0]$ and $x \in [-L, 0] \to [0, L]$, additionally it multiplies every other mode by -1, and it assumes you are working on a 2π periodic domain. These properties are a consequence of the FFT algorithm discussed in detail at a later time.

To see the practical implications of the FFT, we consider the transform of a Gaussian function. The transform can be calculated analytically so that we have the exact relations:

$$f(x) = \exp(-\alpha x^2) \quad \to \quad \widehat{f}(k) = \frac{1}{\sqrt{2\alpha}} \exp\left(-\frac{k^2}{4\alpha}\right). \tag{9.4.1}$$

A simple Python code to verify this with $\alpha = 1$ is as follows

```
L = 20  # define the computational domain [-L/2,L/2]
n = 128  # define the number of Fourier modes 2^n
x2 = np.linspace(-L/2, L/2, n+1) # Define the domain
x = x2[:n] # Consider only the first n points

u = np.exp(-x * x)
ut = np.fft.fft(u)
utshift = np.fft.fftshift(ut)
```

The second figure generated by this script shows how the pulse is shifted. By using the command *fftshift*, we can shift the transformed function back to its mathematically correct positions as shown in the third figure generated. However, before inverting the transformation, it is crucial that the transform is shifted back to the form of the second figure. The command *ifftshift* does this. In general, unless you need to plot the spectrum, it is better not to deal with the *fftshift* and *ifftshift* commands. A graphical representation of the *fft* procedure and its shifting properties is illustrated in Fig. 9.4, where a Gaussian is transformed and shifted by the *fft* routine.

To take a derivative, we need to calculate the k values associated with the transformation. The following example does this. Recall that the FFT assumes a 2π periodic domain that gets

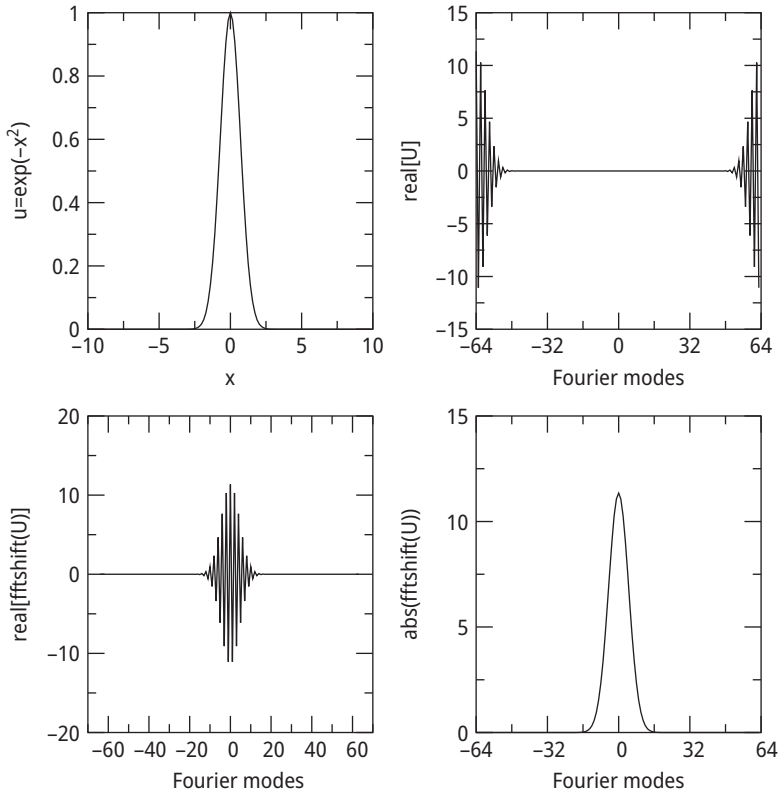


Figure 9.4 Fast Fourier Transform of Gaussian data illustrating the shifting properties of the FFT routine. Note that the *fftshift* command restores the transform to its mathematically correct, unshifted state.

shifted. Thus, the calculation of the k values needs to shift and rescale to the 2π domain. The following example differentiates the function $f(x) = \mathrm{sech(x)}$ three times. The first derivative is compared with the analytic value of $f'(x) = -\mathrm{sech(x)}\tanh(\mathrm{x})$.

```
def sech(x):
    return 1 / np.cosh(x)

def tanh(x):
    return np.sinh(x) / np.cosh(x)

L = 20
n = 128
x2 = np.linspace(-L/2, L/2, n+1)
x = x2[:n]

u = sech(x)
ut = np.fft.fft(u)

k = (2 * np.pi / L) * np.concatenate((np.arange(0, n//2),
  np.arange(-n//2, 0)))
ut1 = 1j * k * ut # first derivative
ut2 = -k**2 * ut # second derivative
ut3 = -1j * k**3 * ut  # third derivative

u1 = np.fft.ifft(ut1) # Inverse transform
u2 = np.fft.ifft(ut2)
u3 = np.fft.ifft(ut3)
```

The routine accounts for the periodic boundaries, the correct k values, and differentiation. Note that no shifting was necessary since we constructed the k values in the shifted space.

For transforming in higher dimensions, a couple of choices in Python are possible. For 2D transformations, it is recommended to use the commands *fft2* and *ifft2*. These will transform a matrix $\mathbf{A}$, which represents data in the x- and y-directions, respectively, along the rows and then columns. For higher dimensions, the *fft* command can be modified to *fft(x,[],N)* where N is the number of dimensions.

9.4.2 Sparse matrices: SPDIAGS, SPY

Under discretization, most physical problems yield sparse matrices, i.e. matrices which are largely composed of zeros. For instance, the matrices (8.6.11) and (9.1.13) are sparse matrices generated under the discretization of a boundary value problem and Poisson equation, respectively. The *spdiag* command allows for the construction of sparse matrices in a relatively simple fashion. The sparse matrix is then saved using a minimal amount of memory and all matrix operations are conducted as usual. The *spy* command allows you to look at the nonzero components of the matrix structure (see Fig. 9.5). As an example, we construct the matrix given by (9.1.13) for the case of $N = 5$ in both the x- and y-directions.

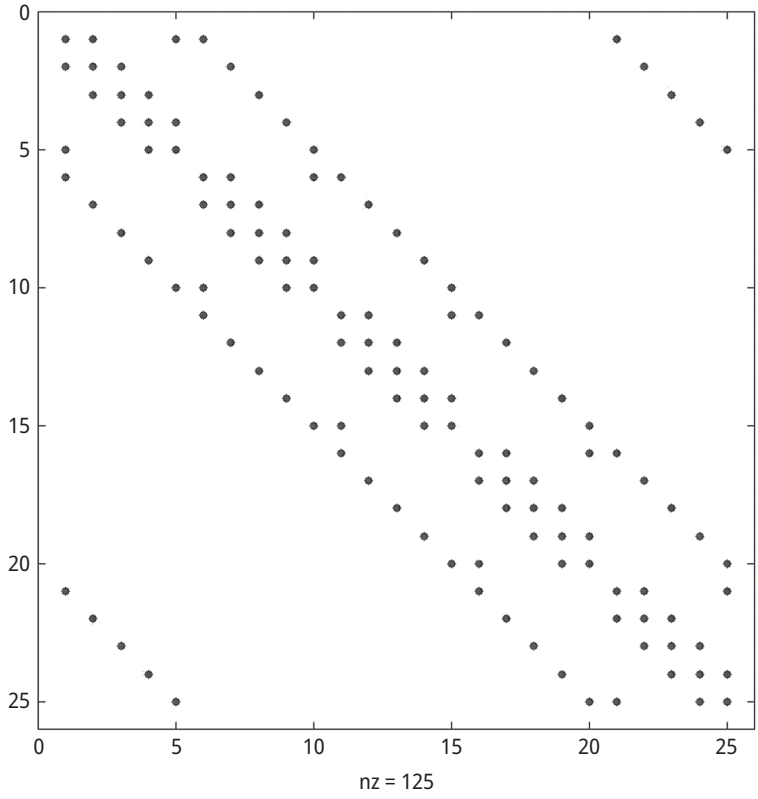


Figure 9.5 Sparse matrix structure for the Laplacian operator using second-order discretization. The output is generated via the *spy* command in MATLAB.

```
from scipy.sparse import spdiags

m = 5    # N value in x and y directions
n = m * m  # total size of matrix

e0 = np.zeros((n, 1))  # vector of zeros
e1 = np.ones((n, 1))   # vector of ones
e2 = np.copy(e1)    # copy the one vector
e4 = np.copy(e0)    # copy the zero vector

for j in range(1, m+1):
    e2[m*j-1] = 0  # overwrite every m^th value with zero
    e4[m*j-1] = 1  # overwirte every m^th value with one

# Shift to correct positions
e3 = np.zeros_like(e2)
e3[1:n] = e2[0:n-1]
e3[0] = e2[n-1]

e5 = np.zeros_like(e4)
e5[1:n] = e4[0:n-1]
e5[0] = e4[n-1]

# Place diagonal elements
diagonals = [e1.flatten(), e1.flatten(), e5.flatten(),
             e2.flatten(), -4 * e1.flatten(), e3.flatten(),
             e4.flatten(), e1.flatten(), e1.flatten()]
offsets = [-(n-m), -m, -m+1, -1, 0, 1, m-1, m, (n-m)]

matA = spdiags(diagonals, offsets, n, n).toarray()
plt.spy(matA)
```

The appropriate construction of the sparse matrix is critical to solving any problem. It is essential to carefully check the matrix for accuracy before using it in an application. As can be seen from this example, it takes a bit of work to get the matrix correct. However, once constructed properly, it can be used with confidence in all other matrix applications and operations.

9.4.3 Iterative methods: BICGSTAB, GMRES

Iterative techniques also need to be considered. There are a wide variety of built-in iterative techniques in Python. Two of the more promising methods are discussed here: the bi-conjugate stabilized gradient method (*bicgstab*) and the generalized minimum residual method (*gmres*). Both are easily implemented in Python.

Recall that these iteration methods are for solving the linear system $\mathbf{Ax} = \mathbf{b}$. The basic call to the generalized minimum residual method is

```
from scipy.sparse.linalg import gmres
x = gmres(A, b)[0]
```

Likewise, the bi-conjugate stabilized gradient method is called by

```
from scipy.sparse.linalg import bicgstab
x, exit_code = bicgstab(A, b)
```

It is rare that you would use these commands without options. The iteration scheme stops upon convergence, failure to converge, or when the maximum number of iterations has been achieved. By default, the initial guess for the iteration procedure is the zero vector. The default number of maximum iterations is 10, which is rarely enough iterations for the purposes of scientific computing.

Thus, it is important to understand the different options available with these iteration techniques. The most general user-specified command line for either *gmres* or *bicgstab* is as follows:

```
x, flag, relres, iter = bicgstab(A, b, tol=tol,
     maxiter=maxit, M1=M1, M2=M2, x0=x0)
x, flag, relres, iter = gmres(A, b, tol=tol, restart=restart,
     maxiter=maxit, M=M1, x0=x0)
```

We already know that the matrix $\mathbf{A}$ and vector $\mathbf{b}$ come from the original problem. The remaining parameters are as follows:

tol $=$ specified tolerance for convergence

maxit $=$ maximum number of iterations

M1, M2 $=$ preconditioning matrices

x0 $=$ initial guess vector

restart $=$ restart of iterations (gmres only).

In addition to these input parameters, *gmres* or *bicgstab* will give the relative residual, *relres*, and the number of iterations performed, *iter*. The *flag* variable gives information on whether the scheme converged in the maximum allowable iterations or not.

9.5 Overcoming Computational Difficulties

In developing algorithms for any given problem, you should always try to maximize the use of information available to you about the specific problem. Specifically, a well-developed numerical code will make extensive use of analytic information concerning the problem. Judicious use of properties of the specific problem can lead to significant reduction in the amount of computational time and effort needed to perform a given calculation.

Here, various practical issues that may arise in the computational implementation of a given method are considered. Analysis provides a strong foundation for understanding the issues, which can be problematic on a computational level. Thus, the focus here will be on details that need to be addressed before straightforward computing is performed.

9.5.1 Streamfunction equations: Nonuniqueness

We return to the consideration of the streamfunction equation

$$\nabla^2\psi = \omega \tag{9.5.1}$$

with the periodic boundary conditions

$$\psi(-L, y, t) = \psi(L, y, t) \tag{9.5.2a}$$

$$\psi(x, -L, t) = \psi(x, L, t)\,. \tag{9.5.2b}$$

The mathematical problem that arises in solving this Poisson equation has been considered previously. Namely, the solution can only be determined to an arbitrary constant. Thus, if ψ_0 is a solution to the streamfunction equation, so is

$$\psi = \psi_0 + c, \tag{9.5.3}$$

where c is an arbitrary constant. This gives an infinite number of solutions to the problem. Thus, upon discretizing the equation in x and y and formulating the matrix formulation of the problem $\mathbf{Ax} = \mathbf{b}$, we find that the matrix $\mathbf{A}$, which is given by (9.1.13), is singular, i.e. $\det \mathbf{A} = 0$.

Obviously, the fact that the matrix $\mathbf{A}$ is singular will create computational problems. However, does this nonuniqueness jeopardize the validity of the physical model? Recall that the streamfunction is a fictitious quantity, which captures the fluid velocity through the quantities $\partial\psi/\partial x$ and $\partial\psi/\partial y$. In particular, we have the x and y velocity components

$$u = -\frac{\partial\psi}{\partial y}\ (x - \text{component of velocity}) \tag{9.5.4a}$$

$$v = \frac{\partial \psi}{\partial x} \, (y - \text{component of velocity}). \tag{9.5.4b}$$

Thus, the arbitrary constant c drops out when calculating physically meaningful quantities. Further, when considering the advection–diffusion equation

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{9.5.5}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}, \tag{9.5.6}$$

only the derivative of the streamfunction is important, which again removes the constant c from the problem formulation.

We can thus conclude that the nonuniqueness from the constant c does not generate any problems for the physical model considered. However, we still have the mathematical problem of dealing with a singular matrix. It should be noted that this problem also occurs when all the boundary conditions are of the Neumann type, i.e. $\partial\psi/\partial n = 0$, where n denotes the outward normal direction.

To overcome this problem numerically, we simply observe that we can arbitrarily add a constant to the solution. Or alternatively, we can pin down the value of the streamfunction at a single location in the computational domain. This constraint fixes the arbitrary constant problem and removes the singularity from the matrix $\mathbf{A}$. Thus to fix the problem, we can simply pick an arbitrary point in our computational domain ψ_{mn} and fix its value. Essentially, this will alter a single component of the sparse matrix (9.1.13). And in fact, this is the simplest thing to do. For instance, given the construction of the matrix (9.1.13), we could simply add the following line of Python code:

```
A[0, 0]=0
```

Then $\det \mathbf{A} \neq 0$ and the matrix can be used in any of the linear solution methods. Note that the choice of the matrix component and its value are completely arbitrary. However, in this example, if you choose to alter this matrix component, to overcome the matrix singularity, you must have $A(1,1) \neq -4$.

9.5.2 Fast Fourier transforms: Divide by zero

In addition to solving the streamfunction equation by standard discretization and $\mathbf{Ax} = \mathbf{b}$, we could use the Fourier transform method. In particular, Fourier transforming in both x and y reduces the streamfunction equation

$$\nabla^2 \psi = \omega \tag{9.5.7}$$

to (9.3.13)

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2}\,. \tag{9.5.8}$$

Here we have denoted the Fourier transform in x as $\widehat{f(x)}$ and the Fourier transform in y as $\widetilde{g(y)}$. The final step is to inverse-transform in x and y to get back to the solution $\psi(x, y)$. It is recommended that the routines *fft2* and *ifft2* be used to perform the transformations. However, *fft* and *ifft* may also be used in loops or with the dimension option set to 2.

An observation concerning (9.5.8) is that there will be a divide by zero when $k_x = k_y = 0$ at the zero mode. Two options are commonly used to overcome this problem. The first is to modify (9.5.8) so that

$$\widetilde{\widehat{\psi}} = -\frac{\widetilde{\widehat{\omega}}}{k_x^2 + k_y^2 + eps} \tag{9.5.9}$$

where eps is the command for generating a machine precision number, which is on the order of $O(10^{-15})$. It essentially adds a round-off to the denominator that removes the divide by zero problem. A second option, which is more highly recommended, is to redefine the $\mathbf{k}_x$ and $\mathbf{k}_y$ vectors associated with the wavenumbers in the x- and y-directions. Specifically, after defining the $\mathbf{k}_x$ and $\mathbf{k}_y$, we could simply add the command line

```
kx[0] = 1e-6
ky[0] = 1e-6
```

The values of $kx[0] = ky[0] = 0$ by default. This would make the values small but finite so that the divide by zero problem is effectively removed with only a small amount of error added to the problem.

9.5.3 Sparse derivative matrices: Advection terms

The sparse matrix (9.1.13) represents the discretization of the Laplacian, which is accurate to second order. However, when calculating the advection terms given by

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}\,, \tag{9.5.10}$$

only the first derivative is required in both x and y. Associated with each of these derivative terms is a sparse matrix, which must be calculated in order to evaluate the advection term.

The calculation of the x derivative will be considered here. The y derivative can be calculated in a similar fashion. Obviously, it is crucial that the sparse matrices representing these operations be correct. Consider then the second-order discretization of

$$\frac{\partial \omega}{\partial x} = \frac{\omega(x + \Delta x, y) - \omega(x - \Delta x, y)}{2\Delta x}\,. \tag{9.5.11}$$

Using the notation developed previously, we define $\omega(x_m, y_n) = \omega_{mn}$. Thus, the discretization yields

$$\frac{\partial \omega_{mn}}{\partial x} = \frac{\omega_{(m+1)n} - \omega_{(m-1)n}}{2\Delta x}\,, \tag{9.5.12}$$

with periodic boundary conditions. The first few terms of this linear system are as follows:

$$\begin{aligned}
\frac{\partial \omega_{11}}{\partial x} &= \frac{\omega_{21} - \omega_{n1}}{2\Delta x} \\
\frac{\partial \omega_{12}}{\partial x} &= \frac{\omega_{22} - \omega_{n2}}{2\Delta x} \\
&\vdots \\
\frac{\partial \omega_{21}}{\partial x} &= \frac{\omega_{31} - \omega_{11}}{2\Delta x} \\
&\vdots
\end{aligned} \tag{9.5.13}$$

From these individual terms, the linear system $\partial \boldsymbol{\omega}/\partial x = \mathbf{B}\boldsymbol{\omega}$ can be constructed. The critical component is the sparse matrix $\mathbf{B}$. Note that we have used the usual definition of the vector $\boldsymbol{\omega}$

$$\boldsymbol{\omega} = \begin{pmatrix} \omega_{11} \\ \omega_{12} \\ \vdots \\ \omega_{1n} \\ \omega_{21} \\ \omega_{22} \\ \vdots \\ \omega_{n(n-1)} \\ \omega_{nn} \end{pmatrix}. \tag{9.5.14}$$

It can be verified then that the sparse matrix $\mathbf{B}$ is given by the matrix

$$\mathbf{B} = \frac{1}{2\Delta x} \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} \\ -\mathbf{I} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & -\mathbf{I} & \mathbf{0} \end{bmatrix}, \tag{9.5.15}$$

where $\mathbf{0}$ is an $n \times n$ zero matrix and $\mathbf{I}$ is an $n \times n$ identity matrix. Recall that the matrix $\mathbf{B}$ is an $n^2 \times n^2$ matrix. The sparse matrix associated with this operation can be constructed with the *spdiag* command. Following the same analysis, the y derivative matrix can also be constructed. If we call this matrix $\mathbf{C}$, then the advection operator can simply be written as

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} = (\mathbf{B}\boldsymbol{\psi})(\mathbf{C}\boldsymbol{\omega}) - (\mathbf{C}\boldsymbol{\psi})(\mathbf{B}\boldsymbol{\omega})\,. \tag{9.5.16}$$

This forms part of the right-hand side of the system of differential equations, which is then solved using a standard time-stepping algorithm.

10 Time- and Space-Stepping Schemes: Method of Lines

10.1 Basic Time-Stepping Schemes 241
- 10.1.1 The ODE system 241
- 10.1.2 Python implementation 242
- 10.1.3 2D Python implementation 243
- 10.1.4 Method of lines 244

10.2 Time-Stepping Schemes: Explicit and Implicit Methods 245
- 10.2.1 Central differencing in time 247
- 10.2.2 Improved accuracy 247
- 10.2.3 Lax–Wendroff 248
- 10.2.4 Backward Euler: Implicit scheme 248
- 10.2.5 MacCormack scheme 249

10.3 Stability Analysis 249
- 10.3.1 von Neumann analysis 250
- 10.3.2 Forward Euler for one-way wave equation 250
- 10.3.3 Backward Euler for one-way wave equation 251
- 10.3.4 Lax–Wendroff for one-way wave equation 252
- 10.3.5 Leap-frog (2,2) for one-way wave equation 252

10.4 Comparison of Time-Stepping Schemes 253
- 10.4.1 Diffusion equation 254
- 10.4.2 Hyper-diffusion 255

10.5 Operator Splitting Techniques 256
- 10.5.1 Operator splitting 257
- 10.5.2 Additional advantages of splitting 257
- 10.5.3 Symmetrized splitting 258

10.6 Optimizing Computational Performance: Rules of Thumb 259
- 10.6.1 One-way wave equation 261
- 10.6.2 Heat equation 262
- 10.6.3 Nonlinear Schrödinger equation 264

With the Fourier transform and discretization in hand, we can turn towards the solution of partial differential equations whose solutions need to be advanced forward in time. Thus, in addition to spatial discretization, we will need to discretize in time in a self-consistent way so as to advance the solution to a desired future time. Issues of stability and accuracy are, of course, at the heart of a discussion on time- and space-stepping schemes.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.
© J. Nathan Kutz (2026). DOI: 10.1093/oso/9780198929093.003.0011

10.1 Basic Time-Stepping Schemes

Basic time-stepping schemes involve discretization in both space and time. Issues of numerical stability as the solution is propagated in time and accuracy from the space–time discretization are of greatest concern for any implementation. To begin this study, we will consider very simple and well-known examples from partial differential equations. The first equation we consider is the heat (diffusion) equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{10.1.1}$$

with the periodic boundary conditions $u(-L,t) = u(L,t)$. We discretize the spatial derivative with a second-order scheme (see Table 10.1) so that

$$\frac{\partial u}{\partial t} = \frac{\kappa}{\Delta x^2}\left[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)\right] . \tag{10.1.2}$$

This approximation reduces the partial differential equation to a system of ordinary differential equations. We have already considered a variety of time-stepping schemes for differential equations and we can apply them directly to this resulting system.

10.1.1 The ODE system

To define the system of ODEs, we discretize and define the values of the vector $\mathbf{u}$ in the following way.

$$\begin{aligned}
&u(-L,t) = u_1\\
&u(-L+\Delta x,t) = u_2\\
&\vdots\\
&u(L-2\Delta x,t) = u_{n-1}\\
&u(L-\Delta x,t) = u_n\\
&u(L,t) = u_{n+1}\,.
\end{aligned}$$

Recall from periodicity that $u_1 = u_{n+1}$. Thus, our system of differential equations solves for the vector

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} . \tag{10.1.3}$$

The governing equation (10.1.2) is then reformulated as the differential equation system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\Delta x^2}\mathbf{A}\mathbf{u}\,, \tag{10.1.4}$$

where $\mathbf{A}$ is given by the sparse matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}, \tag{10.1.5}$$

and the values of 1 on the upper right and lower left of the matrix result from the periodic boundary conditions.

10.1.2 Python implementation

The system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm would be as follows:

(1) Build the sparse matrix $\mathbf{A}$.

```
e1 = np.ones(n) # Build a vector of ones
diagonals = [e1, -2*e1, e1]
offsets = [-1, 0, 1]
A = diags(diagonals, offsets, shape=(n, n), format='csr')
A[0, n-1] = 1 # Periodic boundaries
A[n-1, 0] = 1
```

(2) Generate the desired initial condition vector $\mathbf{u} = \mathbf{u}_0$.

(3) Call an ODE solver from the Python suite. The matrix $\mathbf{A}$, the diffusion constant κ, and spatial step Δx need to be passed into this routine.

```
def rhs(u, t, k, dx, A):   # Define the right-hand side
    return (k / dx**2) * A.dot(u)

u0 = np.exp(-x**2) # Initial condition
y = odeint(rhs, u0, tspan, args=(k, dx, A)) # Solve ODE
```

(4) Plot the results as a function of time and space.

The algorithm is thus fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms already available in Python.

10.1.3 2D Python implementation

In the case of two dimensions, the calculation becomes slightly more difficult since the 2D data are represented by a matrix, and the ODE solvers require a vector input for the initial data. For this case, the governing equation is

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right). \tag{10.1.6}$$

Discretizing in x and y gives a right-hand side, which takes on the form of (7). Provided $\Delta x = \Delta y = \delta$ are the same, the system can be reduced to the linear system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\delta^2} \mathbf{A}\mathbf{u}\,, \tag{10.1.7}$$

where we have arranged the vector $\mathbf{u}$ in a similar fashion to (14) so that

$$\mathbf{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}, \tag{10.1.8}$$

where we have defined $u_{jk} = u(x_j, y_k)$. The matrix $\mathbf{A}$ is a nine diagonal matrix given by (13). The sparse implementation of this matrix is also given previously.

Again, the system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm follows the same course as the 1D case, but extra care is taken in arranging the 2D data into a vector.

(1) Build the sparse matrix $\mathbf{A}$ (13).

(2) Generate the desired initial condition matrix $\mathbf{U} = \mathbf{U}_0$ and reshape it to a vector $\mathbf{u} = \mathbf{u}_0$. This example considers the case of a simple Gaussian as the initial condition. The *reshape* and *meshgrid* commands are important for computational implementation.

```
Lx = 20    # spatial domain of x
Ly = 20    # spatial domain of y
nx = 100   # number of discretization points in x
ny = 100   # number of discretization points in y
N = nx * ny   # elements in reshaped initial condition

x2 = np.linspace(-Lx/2, Lx/2, nx+1) # x domain
x = x2[:nx]
y2 = np.linspace(-Ly/2, Ly/2, ny+1) # y domain
y = y2[:ny]
X, Y = np.meshgrid(x, y)  # make 2D

U = np.exp(-X**2 - Y**2) # Generate a Gaussian matrix
u = U.flatten()[:N].reshape(N, 1) # Reshape into a vector
```

(3) Call an ODE solver from the Python suite. The matrix $\mathbf{B}$, the diffusion constant κ, and spatial step $\Delta x = \Delta y = dx$ need to be passed into this routine.

```
y = odeint(rhs, u0, tspan, args=(k, dx, B)) # Solve ODE
```

The function *rhs.m* should be of the following form:

```
def rhs(u, t, k, dx, B):   # Define the right-hand side
    return (k / dx**2) * B.dot(u)
```

(4) Reshape and plot the results as a function of time and space.

The algorithm is again fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms.

10.1.4 Method of lines

Fundamentally, these methods use the data at a single slice of time to generate a solution Δt in the future. This is then used to generate a solution $2\Delta t$ into the future. The process continues until the desired future time is achieved. This process of solving a partial differential equation is known as the *method of lines*. Each *line* is the value of the solution at a given

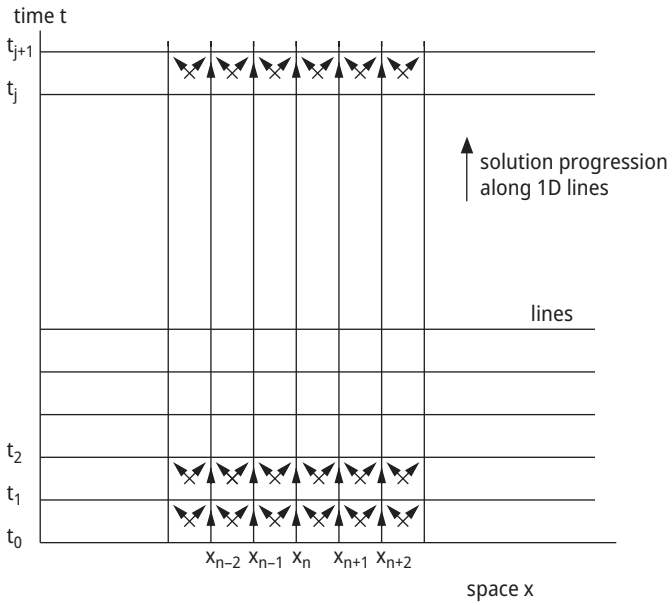


Figure 10.1 Graphical representation of the progression of the numerical solution using the method of lines. Here a second-order discretization scheme is considered, which couples each spatial point with its nearest neighbour.

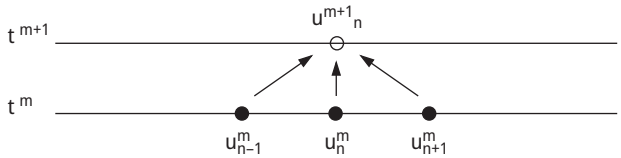


Figure 10.2 Four-point stencil for second-order spatial discretization and Euler time-stepping of the one-wave wave equation.

time slice. The lines are used to update the solution to a new timeline and progressively generate future solutions. Figure 10.1 depicts the process involved in the method of lines for the 1D case. The 2D case was illustrated previously in Fig. 10.2.

10.2 Time-Stepping Schemes: Explicit and Implicit Methods

Now that several technical and computational details have been addressed, we continue to develop methods for time-stepping the solution into the future. Some of the more common schemes will be considered along with a graphical representation of the scheme. Every

scheme eventually leads to an iteration procedure, which the computer can use to advance the solution in time.

We will begin by considering the simplest partial differential equations. Often, it is difficult to do much analysis with complicated equations. Therefore, considering simple equations is not merely an exercise, but rather they are typically the only equations we can make analytical progress with.

As an example, we consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{10.2.1}$$

The simplest discretization of this equation is to first central-difference in the x-direction. This yields

$$\frac{du_n}{dt} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right)\,, \tag{10.2.2}$$

where $u_n = u(x_n, t)$. We can then step forward with an Euler time-stepping method. Denoting $u_n^{(m)} = u(x_n, t_m)$ and applying the method of lines iteration scheme gives

$$u_n^{(m+1)} = u_n^{(m)} + \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right)\,. \tag{10.2.3}$$

This simple scheme has the four-point stencil shown in Fig. 10.2. To illustrate more clearly the iteration procedure, we rewrite the discretized equation in the form

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{10.2.4}$$

where

$$\lambda = \frac{c\Delta t}{\Delta x} \tag{10.2.5}$$

is known as the CFL (Courant–Friedrichs–Lewy) number [41]. The iteration procedure assumes that the solution does not change significantly from one time-step to the next, i.e. $u_n^{(m)} \approx u_n^{(m+1)}$. The accuracy and stability of this scheme is controlled almost exclusively by the CFL number λ. This parameter relates the spatial and time discretization schemes in (5). Note that decreasing Δx without decreasing Δt leads to an increase in λ, which can result in instabilities. Smaller values of Δt suggest smaller values of λ and improved numerical stability properties. In practice, you want to take Δx and Δt as large as possible for computational speed and efficiency without generating instabilities.

There are practical considerations to keep in mind relating to the CFL number. First, given a spatial discretization step-size Δx, you should choose the time discretization so that the CFL number is kept in check. Often a given scheme will only work for CFL conditions below a certain value, thus the importance of choosing a small enough time-step. Second, if indeed, you choose to work with very small Δt, then although stability properties are improved with a lower CFL number, the code will also slow down accordingly. Thus, achieving good stability results is often counter-productive to fast numerical solutions.

10.2.1 Central differencing in time

We can discretize the time-step in a similar fashion to the spatial discretization. Instead of the Euler time-stepping scheme used above, we could central-difference in time using Table 10.1. Thus, after spatial discretization, we have

$$\frac{du_n}{dt} = \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right) , \tag{10.2.6}$$

as before. And using a central difference scheme in time now yields

$$\frac{u_n^{(m+1)} - u_n^{(m-1)}}{2\Delta t} = \frac{c}{2\Delta x}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) . \tag{10.2.7}$$

This last expression can be rearranged to give

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) . \tag{10.2.8}$$

This iterative scheme is called *leap-frog (2,2)* since it is $O(\Delta t^2)$ accurate in time and $O(\Delta x^2)$ accurate in space. It uses a four-point stencil as shown in Fig. 10.3. Note that the solution utilizes two time slices to leap-frog to the next time slice. Thus, the scheme is not self-starting since only one time slice (initial condition) is given.

10.2.2 Improved accuracy

We can improve the accuracy of any of the above schemes by using higher-order central differencing methods. The fourth-order accurate scheme from Table 3.2 gives

$$\frac{\partial u}{\partial x} = \frac{-u(x+2\Delta x) + 8u(x+\Delta x) - 8u(x-\Delta x) + u(x-2\Delta x)}{12\Delta x} . \tag{10.2.9}$$

Combining this fourth-order spatial scheme with second-order central differencing in time gives the iterative scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left[\frac{4}{3}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) - \frac{1}{6}\left(u_{n+2}^{(m)} - u_{n-2}^{(m)}\right)\right] . \tag{10.2.10}$$

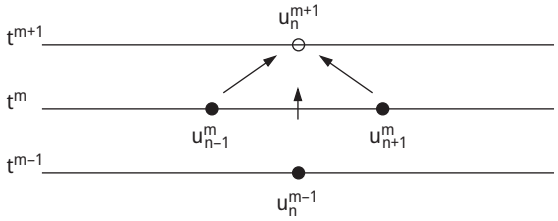


Figure 10.3 Four-point stencil for second-order spatial discretization and central-difference time-stepping of the one-wave wave equation.

This scheme, which is based upon a six-point stencil, is called *leap-frog (2,4)*. It is typical that for *(2,4)* schemes, the maximum CFL number for stable computations is reduced from the basic *(2,2)* scheme.

10.2.3 Lax–Wendroff

Another alternative to discretizing in time and space involves a clever use of the Taylor expansion

$$u(x,t+\Delta t) = u(x,t) + \Delta t \frac{\partial u(x,t)}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial^2 u(x,t)}{\partial t^2} + O(\Delta t^3)\,. \tag{10.2.11}$$

But we note from the governing one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \approx \frac{c}{2\Delta x}\left(u_{n+1} - u_{n-1}\right)\,. \tag{10.2.12a}$$

Taking the derivative of the equation results in the relation

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} \approx \frac{c^2}{\Delta x^2}\left(u_{n+1} - 2u_n + u_{n-1}\right)\,. \tag{10.2.13a}$$

These two expressions for $\partial u/\partial t$ and $\partial^2 u/\partial t^2$ can be substituted into the Taylor series expression to yield the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right)\,. \tag{10.2.14}$$

This iterative scheme is similar to the Euler method. However, it introduces an important *stabilizing* diffusion term, which is proportional to λ^2. This is known as the *Lax–Wendroff* scheme. Although useful for this example, it is difficult to implement in practice for variable coefficient problems. It illustrates, however, the variety and creativity in developing iteration schemes for advancing the solution in time and space.

10.2.4 Backward Euler: Implicit scheme

The backward Euler method uses the future time for discretizing the spatial domain. Thus, upon discretizing in space and time, we arrive at the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right)\,. \tag{10.2.15}$$

This gives the tridiagonal system

$$u_n^{(m)} = -\frac{\lambda}{2}u_{n+1}^{(m+1)} + u_n^{(m+1)} + \frac{\lambda}{2}u_{n-1}^{(m+1)}\,, \tag{10.2.16}$$

which can be written in matrix form as

$$\mathbf{A}\mathbf{u}^{(m+1)} = \mathbf{u}^{(m)} \tag{10.2.17}$$

Table 10.1 Stability of time-stepping schemes as a function of the CFL number.

Scheme	Stability
Forward Euler	unstable for all λ
Backward Euler	stable for all λ
Leap-frog (2,2)	stable for $\lambda \leq 1$
Leap-frog (2,4)	stable for $\lambda \leq 0.707$

where

$$\mathbf{A} = \frac{1}{2}\begin{pmatrix} 2 & -\lambda & \cdots & 0 \\ \lambda & 2 & -\lambda & \cdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \lambda & 2 \end{pmatrix}. \tag{10.2.18}$$

Thus, before stepping forward in time, we must solve a matrix problem. This can severely affect the computational time of a given scheme. The only thing which may make this method viable is if the CFL condition is such that much larger time-steps are allowed, thus overcoming the limitations imposed by the matrix solve.

10.2.5 MacCormack scheme

In the MacCormack scheme, the variable coefficient problem of the Lax–Wendroff scheme and the matrix solve associated with the backward Euler are circumvented by using a predictor–corrector method. The computation thus occurs in two pieces:

$$u_n^{(P)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{10.2.19a}$$

$$u_n^{(m+1)} = \frac{1}{2}\left[u_n^{(m)} + u_n^{(P)} + \lambda\left(u_{n+1}^{(P)} - u_{n-1}^{(P)}\right)\right]. \tag{10.2.19b}$$

This method essentially combines forward and backward Euler schemes so that we avoid the matrix solve and the variable coefficient problem.

The CFL condition will be discussed in detail in the next section. For now, the basic stability properties of many of the schemes considered here are given in Table 10.1. The stability of the various schemes holds only for the one-way wave equation considered here as a prototypical example. Each partial differential equation needs to be considered and classified individually with regards to stability.

10.3 **Stability Analysis**

In the preceding section, we considered a variety of schemes, which can solve the time–space problem posed by partial differential equations. However, it remains undetermined

which scheme is best for implementation purposes. Two criteria provide a basis for judgement: accuracy and stability. For each scheme, we already know the accuracy due to the discretization error. However, a determination of stability is still required.

To start to understand stability, we once again consider the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}\,. \tag{10.3.1}$$

Using Euler time-stepping and central differencing in space gives the iteration procedure

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{10.3.2}$$

where λ is the CFL number.

10.3.1 von Neumann analysis

To determine the stability of a given scheme, we perform a *von Neumann analysis* [42]. This assumes the solution is of the form

$$u_n^{(m)} = g^m \exp(i\xi_n h) \quad \xi \in [-\pi/h, \pi/h] \tag{10.3.3}$$

where $h = \Delta x$ is the spatial discretization parameter. Essentially, this assumes the solution can be constructed of Fourier modes. The key then is to determine what happens to g^m as $m \to \infty$. Two possibilities exist

$$\lim_{m\to\infty} |g|^m \to \infty \qquad \text{unstable scheme} \tag{10.3.4a}$$

$$\lim_{m\to\infty} |g|^m \leq 1 \ (< \infty) \qquad \text{stable scheme.} \tag{10.3.4b}$$

Thus, this stability check is very much like that performed for the time-stepping schemes developed for ordinary differential equations.

10.3.2 Forward Euler for one-way wave equation

The Euler discretization of the one-way wave equation produces the iterative scheme (10.3.2). Plugging in the ansatz (10.3.3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^m \exp(i\xi_{n+1}h) - g^m \exp(i\xi_{n-1}h)\right) \\ g^{m+1}\exp(i\xi_n h) &= g^m\left(\exp(i\xi_n h) + \frac{\lambda}{2}\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1} - \xi_n)) - \exp(ih(\xi_{n-1} - \xi_n))\right]\,. \end{aligned} \tag{10.3.5}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda \sin(\zeta h)\,. \end{aligned} \tag{10.3.6}$$

From this, we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta)g(\zeta)} = \sqrt{1 + \lambda^2 \sin^2 \zeta h}\,. \tag{10.3.7}$$

Thus,

$$|g(\zeta)| \geq 1 \quad \rightarrow \quad \lim_{m\to\infty} |g|^m \rightarrow \infty \quad \text{unstable scheme.} \tag{10.3.8}$$

Thus, the forward Euler time-stepping scheme is unstable for any value of λ. The implementation of this scheme will force the solution to infinity due to numerical instability.

10.3.3 Backward Euler for one-way wave equation

The Euler discretization of the one-way wave equation produces the iterative scheme

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m+1)} - u_{n-1}^{(m+1)}\right)\,. \tag{10.3.9}$$

Plugging in the ansatz (10.3.3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2}\left(g^{m+1}\exp(i\xi_{n+1}h) - g^{m+1}\exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2} g\left[\exp(ih(\xi_{n+1} - \xi_n)) - \exp(ih(\xi_{n-1} - \xi_n))\right]\,. \end{aligned} \tag{10.3.10}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2} g[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g(\zeta) &= 1 + i\lambda g \sin(\zeta h) \\ g[1 - i\lambda \sin(\zeta h)] &= 1 \\ g(\zeta) &= \frac{1}{1 - i\lambda \sin \zeta h}\,. \end{aligned} \tag{10.3.11}$$

From this, we can deduce that

$$|g(\zeta)| = \sqrt{g^*(\zeta)g(\zeta)} = \sqrt{\frac{1}{1 + \lambda^2 \sin^2 \zeta h}}\,. \tag{10.3.12}$$

Thus,

$$|g(\zeta)| \leq 1 \quad \rightarrow \quad \lim_{m\to\infty} |g|^m \leq 1 \quad \text{unconditionally stable.} \tag{10.3.13}$$

Thus, the backward Euler time-stepping scheme is stable for any value of λ. The implementation of this scheme will not force the solution to infinity.

10.3.4 Lax–Wendroff for one-way wave equation

The Lax–Wendroff scheme is not as transparent as the forward and backward Euler schemes.

$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) + \frac{\lambda^2}{2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right). \tag{10.3.14}$$

Plugging in the standard ansatz (10.3.3) gives

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^m \exp(i\xi_n h) + \frac{\lambda}{2} g^m \left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right) \\ &\quad + \frac{\lambda^2}{2} g^m \left(\exp(i\xi_{n+1}h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1}h)\right) \\ g &= 1 + \frac{\lambda}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right] \\ &\quad + \frac{\lambda^2}{2}\left[\exp(ih(\xi_{n+1}-\xi_n)) + \exp(ih(\xi_{n-1}-\xi_n)) - 2\right]. \end{aligned} \tag{10.3.15}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g(\zeta) &= 1 + \frac{\lambda}{2}[\exp(i\zeta h) - \exp(-i\zeta h)] + \frac{\lambda^2}{2}[\exp(i\zeta h) + \exp(-i\zeta h) - 2] \\ g(\zeta) &= 1 + i\lambda \sin\zeta h + \lambda^2(\cos\zeta h - 1) \\ g(\zeta) &= 1 + i\lambda \sin\zeta h - 2\lambda^2 \sin^2(\zeta h/2). \end{aligned} \tag{10.3.16}$$

This results in the relation

$$|g(\zeta)|^2 = \lambda^4\left[4\sin^4(\zeta h/2)\right] + \lambda^2\left[\sin^2\zeta h - 4\sin^2(\zeta h/2)\right] + 1. \tag{10.3.17}$$

This expression determines the range of values for the CFL number λ for which $|g| \leq 1$. Ultimately, the stability of the Lax–Wendroff scheme for the one-way wave equation is determined.

10.3.5 Leap-frog (2,2) for one-way wave equation

The leap-frog discretization for the one-way wave equation yields the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right). \tag{10.3.18}$$

Plugging in the ansatz (10.3.3) gives the following relations:

$$\begin{aligned} g^{m+1}\exp(i\xi_n h) &= g^{m-1}\exp(i\xi_n h) + \lambda g^m\left(\exp(i\xi_{n+1}h) - \exp(i\xi_{n-1}h)\right) \\ g^2 &= 1 + \lambda g\left[\exp(ih(\xi_{n+1}-\xi_n)) - \exp(ih(\xi_{n-1}-\xi_n))\right]. \end{aligned} \tag{10.3.19}$$

Letting $\xi_n = n\zeta$ reduces the equations further to

$$\begin{aligned} g^2 &= 1 + \lambda g[\exp(i\zeta h) - \exp(-i\zeta h)] \\ g - \frac{1}{g} &= 2i\lambda\sin(\zeta h). \end{aligned} \tag{10.3.20}$$

To assure stability, it can be shown that we require

$$\lambda \leq 1. \tag{10.3.21}$$

Thus, the leap-frog (2,2) time-stepping scheme is stable for values of $\lambda \leq 1$. The implementation of this scheme with this restriction will not force the solution to infinity.

A couple of general remarks should be made concerning the von Neumann analysis.

- It is a general result that a scheme, which is forward in time and centred in space is unstable for the one-way wave equation. This assumes a standard forward discretization, not something like Runge–Kutta.
- von Neumann analysis is rarely enough to guarantee stability, i.e. it is necessary but not sufficient.
- Many other mechanisms for unstable growth are not captured by von Neumann analysis.
- Nonlinearity usually kills the von Neumann analysis immediately. Thus, a large variety of nonlinear partial differential equations are beyond the scope of a von Neumann analysis.
- Accuracy versus stability: It is always better to worry about accuracy. An unstable scheme will quickly become apparent by causing the solution to blow-up to infinity, whereas an inaccurate scheme will simply give you a wrong result without indicating a problem.

10.4 Comparison of Time-Stepping Schemes

A few open questions remain concerning the stability and accuracy issues of the time- and space-stepping schemes developed. In particular, it is not clear how the stability results derived in the previous section apply to other partial differential equations.

Consider, for instance, the leap-frog (2,2) method applied to the one-way wave equation

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x}. \tag{10.4.1}$$

The leap-frog discretization for the one-way wave equation yielded the iteration scheme

$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda \left(u_{n+1}^{(m)} - u_{n-1}^{(m)} \right) , \tag{10.4.2}$$

where $\lambda = c\Delta t/\Delta x$ is the CFL number. The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$g - \frac{1}{g} = 2i\lambda \sin(\zeta h) , \tag{10.4.3}$$

where $\xi_n = n\zeta$. Thus, the scheme was stable provided $\lambda \leq 1$. Note that to double the accuracy, both the time and space discretizations Δt and Δx need to be simultaneously halved.

10.4.1 Diffusion equation

We will again consider the leap-frog (2,2) discretization applied to the diffusion equation. The stability properties will be found to be very different from those of the one-way wave equation. The difference in the one-way wave equation and diffusion equation is an extra x derivative so that

$$\frac{\partial u}{\partial t} = c \frac{\partial^2 u}{\partial x^2} . \tag{10.4.4}$$

Discretizing the spatial second derivative yields

$$\frac{\partial u_n^{(m)}}{\partial t} = \frac{c}{\Delta x^2} \left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)} \right) . \tag{10.4.5}$$

Second-order centre-differencing in time then yields

$$u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda \left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)} \right) , \tag{10.4.6}$$

where now the CFL number is given by

$$\lambda = \frac{c\Delta t}{\Delta x^2} . \tag{10.4.7}$$

In this case, to double the accuracy and hold the CFL number constant requires cutting the time-step by a factor of 4. This is generally true of any partial differential equation with the highest derivative term having two derivatives. The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$\begin{aligned} g^{m+1} \exp(i\xi_n h) &= g^{m-1} \exp(i\xi_n h) + 2\lambda g^m \left(\exp(i\xi_{n+1} h) - 2\exp(i\xi_n h) + \exp(i\xi_{n-1} h) \right) \\ g - \frac{1}{g} &= 2\lambda (\exp(i\zeta h) + \exp(-i\zeta h) - 2) \\ g - \frac{1}{g} &= 2\lambda (i \sin(\zeta h) - 1) , \end{aligned} \tag{10.4.8}$$

where we let $\xi_n = n\zeta$. Unlike the one-way wave equation, the addition of the term $-2u_n^{(m)}$ in the discretization makes $|g(\zeta)| \geq 1$ for all values of λ. Thus, the leap-frog (2,2) is unstable for all CFL numbers for the diffusion equation.

Interestingly enough, the forward Euler method which was unstable when applied to the one-way wave equation can be stable for the diffusion equation. Using an Euler discretization instead of a centre-difference scheme in time yields

$$u_n^{(m+1)} = u_n^{(m)} + \lambda \left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)} \right) . \tag{10.4.9}$$

The von Neumann stability analysis based upon the ansatz $u_n^{(m)} = g^m \exp(i\xi_n h)$ results in the expression

$$g = 1 + \lambda(i \sin(\zeta h) - 1) \tag{10.4.10}$$

This gives

$$|g| = 1 - 2\lambda + \lambda^2(1 + \sin^2 \zeta h) , \tag{10.4.11}$$

so that

$$|g| \leq 1 \quad \text{for} \quad \lambda \leq \frac{1}{2} . \tag{10.4.12}$$

Thus, the maximum step-size in time is given by $\Delta t = \Delta x^2/2c$. Again, it is clear that to double accuracy, the time-step must be reduced by a factor of 4.

10.4.2 Hyper-diffusion

Higher derivatives in x require central differencing schemes, which successively add powers of Δx to the denominator. This makes for severe time-stepping restrictions in the CFL number. As a simple example, consider the fourth-order diffusion equation

$$\frac{\partial u}{\partial t} = -c \frac{\partial^4 u}{\partial x^4} . \tag{10.4.13}$$

Using a forward Euler method in time and a central difference scheme in space gives the iteration scheme

$$u_n^{(m+1)} = u_n^{(m)} - \lambda \left(u_{n+2}^{(m)} - 4u_{n+1}^{(m)} + 6u_n^{(m)} - 4u_{n-1}^{(m)} + u_{n-2}^{(m)} \right) , \tag{10.4.14}$$

where the CFL number λ is now given by

$$\lambda = \frac{c\Delta t}{\Delta x^4} . \tag{10.4.15}$$

Thus, doubling the accuracy requires a drop in the step-size to $\Delta t/16$, i.e. the run time takes 32 times as long since there are twice as many spatial points and 16 times as many time-steps. This kind of behaviour is often referred to as numerical stiffness.

Numerical stiffness of this sort is not a result of the central differencing scheme. Rather, it is an inherent problem with hyper-diffusion. For instance, we could consider solving the fourth-order diffusion equation (10.4.13) with fast Fourier transforms. Transforming the equation in the x-direction gives

$$\frac{\partial \widehat{u}}{\partial t} = -c(ik)^4 \widehat{u} = -ck^4 \widehat{u}\,. \tag{10.4.16}$$

This can then be solved with a differential equation time-stepping routine. The time-step of any of the standard time-stepping routines is based upon the size of the right-hand side of the equation. If there are $n = 128$ Fourier modes, then $k_{\max} = 64$. But note that

$$(k_{\max})^4 = (64)^4 = 16.8 \times 10^6\,, \tag{10.4.17}$$

which is a very large value. The time-stepping algorithm will have to adjust to these large values, which are generated strictly from the physical effect of the higher-order diffusion.

There are a few key issues in dealing with numerical stiffness.

- Use a variable time-stepping routine that uses smaller steps when necessary but large steps if possible. Every built-in Python differential equation solver uses an adaptive stepping routine to advance the solution.
- If numerical stiffness is a problem, then it is often useful to use an implicit scheme. This will generally allow for larger time-steps. The time-stepping algorithm *ode113* uses a predictor–corrector method, which partially utilizes an implicit scheme.
- If stiffness comes from the behaviour of the solution itself, i.e. you are considering a singular problem, then it is advantageous to use a solver specifically built for this stiffness. The time-stepping algorithm *ode15s* relies on Gear methods and is well suited to this type of problem.
- From a practical viewpoint, beware of the accuracy of *ode23* or *ode45* when strong nonlinearity or singular behaviour is expected.

10.5 Operator Splitting Techniques

With either the finite difference or spectral methods, the governing partial differential equations are transformed into a system of differential equations, which are advanced in time with any of the standard time-stepping schemes. A von Neumann analysis can often suggest the appropriateness of a scheme. For instance, we have the following:

A.: **wave behaviour**: $\partial u/\partial t = \partial u/\partial x$
- forward Euler: unstable for all λ
- leap-frog (2,2): stable $\lambda \leq 1$

B.: **diffusion behaviour**: $\partial u/\partial t = \partial^2 u/\partial x^2$
- forward Euler: stable $\lambda \leq 1/2$
- leap-frog (2,2): unstable for all λ

Thus, the physical behaviour of the governing equation dictates the kind of scheme, which must be implemented. But what if we wanted to consider the equation

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}? \tag{10.5.1}$$

This has elements of both diffusion and wave propagation. What time-stepping scheme is appropriate for such an equation? Certainly the Euler method seems to work well for diffusion but destabilizes wave propagation. In contrast, leap-frog (2,2) works well for wave behaviour and destabilizes diffusion.

10.5.1 Operator splitting

The key idea behind operator splitting is to decouple the various physical effects of the problem from each other [43]. Over very small time-steps, for instance, one can imagine that diffusion would essentially act independently of the wave propagation and vice versa in (10.5.1). Just as in (10.5.1), the advection–diffusion can also be thought of as decoupling. So we can consider

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{10.5.2}$$

where the bracketed terms represent the advection (wave propagation) and the right-hand side represents the diffusion. Again, there is a combination of wave dynamics and diffusive spreading.

Over a very small time interval Δt, it is reasonable to conjecture that the diffusion process is independent of the advection. Thus, we could split the calculation into the following two pieces:

$$\Delta t: \quad \frac{\partial \omega}{\partial t} + [\psi, \omega] = 0 \quad \text{advection only} \tag{10.5.3a}$$

$$\Delta t: \quad \frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega \quad \text{diffusion only.} \tag{10.5.3b}$$

This then allows us to time-step each physical effect independently over Δt. Advantage can then be taken of an appropriate time-stepping scheme that is stable for those particular terms. For instance, in this decoupling, we could solve the advection (wave propagation) terms with a leap-frog (2,2) scheme and the diffusion terms with a forward Euler method.

10.5.2 Additional advantages of splitting

There can be additional advantages to the splitting scheme. To see this, we consider the nonlinear Schrödinger equation

$$i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u = 0 \tag{10.5.4}$$

where we split the operations so that

$$\mathbf{I}.\ \Delta t: \quad i\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial^2 u}{\partial x^2} = 0 \tag{10.5.5a}$$

$$\mathbf{II}.\ \Delta t: \quad i\frac{\partial u}{\partial t} + |u|^2 u = 0. \tag{10.5.5b}$$

Thus, the solution will be decomposed into a linear and nonlinear part. We begin by solving the linear part **I.** Fourier transforming yields

$$i\frac{d\widehat{u}}{dt} - \frac{k^2}{2}\widehat{u} = 0 \tag{10.5.6}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp\left(-i\frac{k^2}{2}t\right). \tag{10.5.7}$$

So at each step, we simply need to calculate the transform of the initial condition $\widehat{u}_0$, multiply by $\exp(-ik^2t/2)$, and then invert.

The next step is to solve the nonlinear evolution **II** over a time-step Δt. This is easily done since the nonlinear evolution admits the exact solution

$$u = u_0 \exp\left(i|u_0|^2 t\right) \tag{10.5.8}$$

where u_0 is the initial condition. This part of the calculation then requires no computation, i.e. we have an exact, analytic solution. The fact that we can take advantage of this property is why the splitting algorithm is so powerful.

To summarize then, the split-step method for the nonlinear Schrödinger equation yields the following algorithm.

(1) Dispersion: $u_1 = \text{FFT}^{-1}\left[\widehat{u}_0 \exp(-ik^2\Delta t/2)\right]$

(2) Nonlinearity: $u_2 = u_1 \exp(i|u_1|^2\Delta t)$

(3) Solution: $u(t + \Delta t) = u_2$.

Note the advantage that is taken of the analytic properties of the solution of each aspect of the split operators.

10.5.3 Symmetrized splitting

Although we will not perform an error analysis on this scheme, it is not hard to see that the error will depend heavily on the time-step Δt. Thus, our scheme for solving

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{10.5.9}$$

involves the splitting

$$\mathbf{I}.\ \Delta t: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{10.5.10a}$$

$$\mathbf{II}.\ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0\,. \tag{10.5.10b}$$

To drop the error down by another $O(\Delta t)$, we use what is called a Strang splitting technique [44], which is based on the Trotter product formula. This essentially involves symmetrizing the splitting algorithm so that

$$\mathbf{I}.\ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0 \tag{10.5.11a}$$

$$\mathbf{II}.\ \Delta t: \quad \frac{\partial u}{\partial t} + N(u) = 0 \tag{10.5.11b}$$

$$\mathbf{III}.\ \frac{\Delta t}{2}: \quad \frac{\partial u}{\partial t} + Lu = 0. \tag{10.5.11c}$$

This essentially cuts the time-step in half and allows the error to drop down an order of magnitude. It should always be used so as to keep the error in check.

10.6 Optimizing Computational Performance: Rules of Thumb

Computational performance is always crucial in choosing a numerical scheme. Speed, accuracy, and stability all play key roles in determining the appropriate choice of method for solving. We will consider three prototypical equations:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \qquad \text{one-way wave equation} \tag{10.6.1a}$$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \qquad \text{diffusion equation} \tag{10.6.1b}$$

$$i\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + |u|^2 u \qquad \text{nonlinear Schrödinger equation.} \tag{10.6.1c}$$

Periodic boundary conditions will be assumed in each case. The purpose of this section is to build a numerical routine, which will solve these problems using the iteration procedures outlined in the previous two sections. Of specific interest will be the setting of the CFL number and the consequences of violating the stability criteria associated with it.

The equations are considered in one dimension such that the first and second derivative are given by

$$\frac{\partial u}{\partial x} \to \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & -1 & 0 & 1 \\ 1 & 0 & \cdots & 0 & -1 & 0 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \tag{10.6.2}$$

and

$$\frac{\partial^2 u}{\partial x^2} \to \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \tag{10.6.3}$$

From the previous sections, we have the following discretization schemes for the one-way wave equation (10.6.4a):

Euler (unstable):
$$u_n^{(m+1)} = u_n^{(m)} + \frac{\lambda}{2}\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{10.6.4a}$$

leap-frog (2,2) (stable for $\lambda \leq 1$):
$$u_n^{(m+1)} = u_n^{(m-1)} + \lambda\left(u_{n+1}^{(m)} - u_{n-1}^{(m)}\right) \tag{10.6.4b}$$

where the CFL number is given by $\lambda = \Delta t/\Delta x$. Similarly for the diffusion equation (10.6.1b)

Euler (stable for $\lambda \leq 1/2$):
$$u_n^{(m+1)} = u_n^{(m)} + \lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{10.6.5a}$$

leap-frog (2,2) (unstable):
$$u_n^{(m+1)} = u_n^{(m-1)} + 2\lambda\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) \tag{10.6.5b}$$

where now the CFL number is given by $\lambda = \Delta t/\Delta x^2$. The nonlinear Schrödinger equation discretizes to the following form:

$$\frac{\partial u_n^{(m)}}{\partial t} = -\frac{i}{2\Delta x^2}\left(u_{n+1}^{(m)} - 2u_n^{(m)} + u_{n-1}^{(m)}\right) - i|u_n^{(m)}|^2 u_n^{(m)}. \tag{10.6.6}$$

We will explore Euler and leap-frog (2,2) time-stepping with this equation.

10.6.1 One-way wave equation

We first consider the leap-frog (2,2) scheme applied to the one-way wave equation. Figure 10.4 depicts the evolution of an initial Gaussian pulse. For this case, the CFL is 0.5 so that stable evolution is analytically predicted. The solution propagates to the left as expected from the exact solution. The leap-frog (2,2) scheme becomes unstable for $\lambda \geq 1$, and the system is always unstable for the Euler time-stepping scheme. Figure 10.5 depicts the unstable evolution of the leap-frog (2,2) scheme with CFL $= 2$ and the Euler time-stepping scheme. The initial conditions used are identical to that in Fig. 10.4. Since we have predicted that the leap-frog numerical scheme is only stable provided $\lambda < 1$, it is not surprising that the

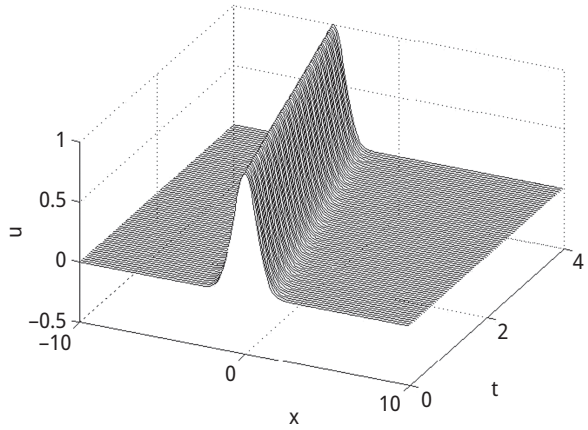


Figure 10.4 Evolution of the one-way wave equation with the leap-frog (2,2) scheme and with CFL=0.5. The stable travelling wave solution is propagated in this case to the left.

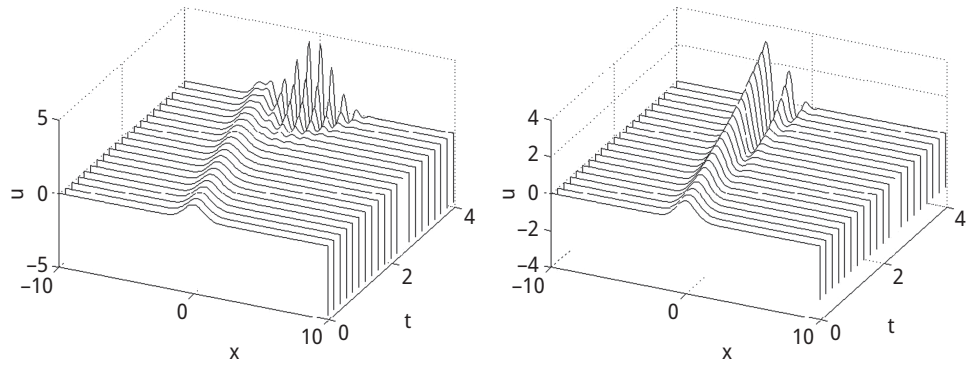


Figure 10.5 Evolution of the one-way wave equation using the leap-frog (2,2) scheme with CFL=2 (left) along with the Euler time-stepping scheme (right). The analysis predicts stable evolution of leap-frog provided the CFL≤ 1. Thus, the onset of numerical instability near $t \approx 3$ for the CFL=2 case is not surprising. Likewise, the Euler scheme is expected to be unstable for all CFL.

figure on the left goes unstable. Likewise, the figure on the right shows the numerical instability generated in the Euler scheme. Note that both of these unstable evolutions develop high-frequency oscillations, which eventually blow up. The Python code used to generate the leap-frog and Euler iterative solutions is given by

```
Time = 4   # time domain of solution
L = 20   # domain size
n = 200   # spatial discretization points
x2 = np.linspace(-L/2, L/2, n+1)
x = x2[:n]
dx = x[1] - x[0]
dt = 0.2
CFL = dt / dx
time_steps = int(Time / dt)
t = np.arange(0, Time + dt, dt)

u0 = np.exp(-x**2) # Initial conditions
u1 = np.exp(-(x + dt)**2)
usol = np.zeros((n, time_steps + 1))
usol[:, 0] = u0
usol[:, 1] = u1

e1 = np.ones(n)  # Sparse matrix for derivative
A = diags([-e1, e1], [-1, 1], shape=(n, n)).toarray()
A[0, -1] = -1
A[-1, 0] = 1

for j in range(time_steps-1):  # Leap frog (2,2)
    u2 = u0 + CFL * A.dot(u1)  # Leap frog (2,2)
    u0 = u1
    u1 = u2
    usol[:, j + 2] = u2
```

10.6.2 Heat equation

In a similar fashion, we investigate the evolution of the diffusion equation when the space–time discretization is given by the leap-frog (2,2) scheme or Euler stepping. Figure 10.6 shows the expected diffusion behaviour for the stable Euler scheme ($\lambda \leq 0.5$). In contrast, Fig. 10.7 shows the numerical instabilities, which are generated from violating the CFL constraint

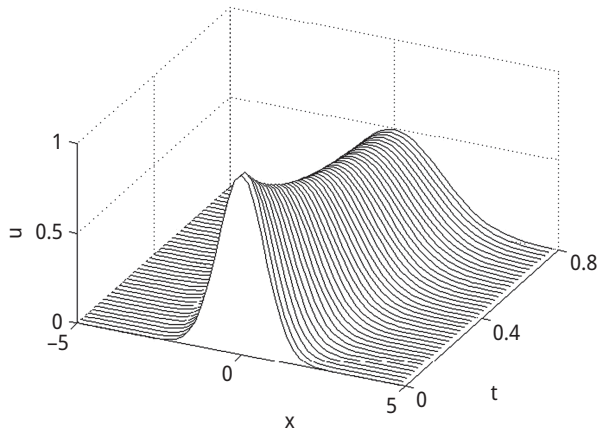


Figure 10.6 Stable evolution of the heat equation with the Euler scheme with CFL=0.5. The initial Gaussian is diffused in this case.

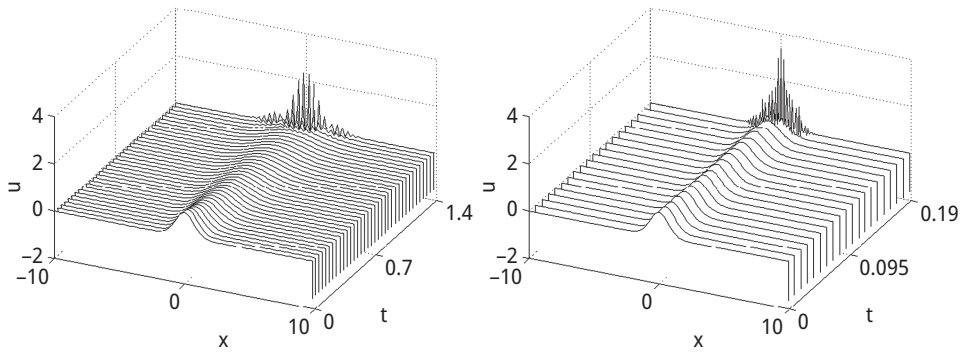


Figure 10.7 Evolution of the heat equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=1. The analysis predicts that both these schemes are unstable. Thus, the onset of numerical instability is observed.

for the Euler scheme or using the always unstable leap-frog (2,2) scheme for the diffusion equation. The numerical code used to generate these solutions follows that given previously for the one-way wave equation. However, the sparse matrix is now given by

```
e1 = np.ones(n)
A = diags([e1, -2*e1, e1], [-1, 0, 1], shape=(n, n)).toarray()
A[0, -1] = 1
A[-1, 0] = 1
```

Further, the iterative process is now

```
# Leap frog (2,2) iteration scheme
for j in range(time_steps - 1):
    u2 = u0 + 2 * CFL * A.dot(u1)  # Leap frog (2,2)
    u0 = u1
    u1 = u2
    usol[:, j + 2] = u2
```

where we recall that the CFL condition is now given by $\lambda = \Delta t/\Delta x^2$, i.e.

```
CFL=dt/dx/dx
```

This solves the one-dimensional heat equation with periodic boundary conditions.

10.6.3 Nonlinear Schrödinger equation

The nonlinear Schrödinger equation can easily be discretized by the above techniques. However, as with most nonlinear equations, it is a bit more difficult to perform a von Neumann analysis. Therefore, we explore the behaviour for this system for two different discretization schemes: Euler and leap-frog (2,2). The CFL number will be the same with both schemes ($\lambda = 0.05$) and the stability will be investigated through numerical computations. Figure 10.8 shows the evolution of the exact one-soliton solution of the nonlinear Schrödinger equation ($u(x, 0) = \text{sech}(x)$) over six units of time. The Euler scheme is observed to lead to numerical instability, whereas the leap-frog (2,2) scheme is stable. In general, the leap-frog schemes work well for wave propagation problems, while Euler methods are better for problems of a diffusive nature.

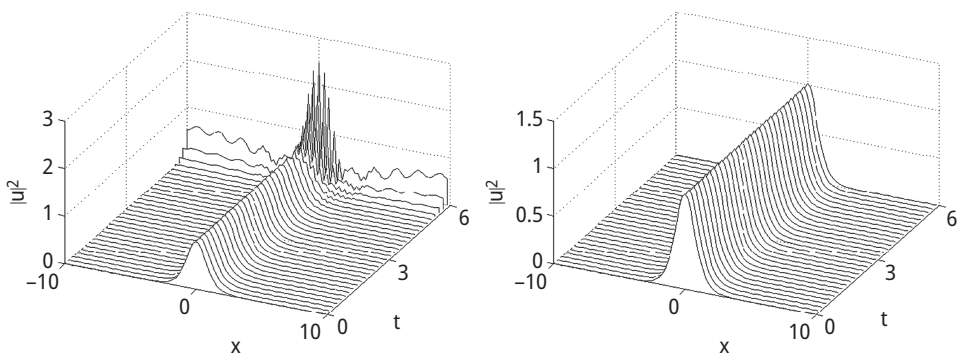


Figure 10.8 Evolution of the nonlinear Schrödinger equation with the Euler time-stepping scheme (left) and leap-frog (2,2) scheme (right) with CFL=0.05.

The Python code modifications necessary to solve the nonlinear Schrödinger equation are trivial. Specifically, the iteration scheme requires change. For the stable leap-frog scheme, the following command structure is required

```
u2 = u0 + -1j * CFL * A.dot(u1) - 1j * 2 * dt * (np.conj(u1) * u1) * u1
u0 = u1; u1 = u2
```

Note that i is automatically defined in Python as $i = \sqrt{-1}$. Thus, it is imperative that you do not use the variable i as a counter in your FOR loops. You will solve a very different equation if you are not careful with this definition.

11 Spectral Methods

11.1 Fast Fourier Transforms and Cosine/Sine Transform 267
11.1.1 Fast Fourier transforms: Cooley–Tukey algorithm 269
11.2 Chebyshev Polynomials and Transform 270
11.2.1 Chebyshev polynomials 271
11.2.2 Solving differential equations 274
11.3 Spectral Method Implementation 274
11.4 Pseudo-Spectral Techniques with Filtering 277
11.4.1 Filtered pseudo-spectral 278
11.4.2 Example: Fisher–Kolmogorov equation 279
11.4.3 Comparison of Spectral and Finite Difference Methods 280
11.5 Boundary Conditions and the Chebyshev Transform 282
11.5.1 Method 1: Periodic extension with FFTs 282
11.5.2 Method 2: Polynomial approximation with equi-spaced points 282
11.5.3 Method 3: Polynomial approximation with clustered points 283
11.5.4 Clustered points and Chebyshev differentiation 284
11.5.5 Boundaries 285
11.5.6 Connecting to the FFT 286
11.6 Implementing the Chebyshev Transform 286
11.6.1 Differentiation matrix in 2D 288
11.6.2 Solving PDEs with the Chebyshev differentiation matrix 289
11.7 Computing Spectra: The Floquet–Fourier–Hill Method 291
11.7.1 Example: Schrödinger operators 293
11.7.2 Example: Mathieu's equation 296

Spectral methods are one of the most powerful solution techniques for ordinary and partial differential equations. The best-known example of a spectral method is the Fourier transform. We have already made use of the Fourier transform using FFT routines. Other spectral techniques exist which render a variety of problems easily tractable and often at significant computational savings.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0012

11.1 Fast Fourier Transforms and Cosine/Sine Transform

From our previous chapters, we are already familiar with the Fourier transform and some of its properties. At that time, the distinct advantage of using the FFT was its computational efficiency in solving a problem in $O(N \log N)$. This section explores the underlying mathematical reasons for such performance.

One of the abstract definitions of a Fourier transform pair is given by

$$F(k) = \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{11.1.1a}$$

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ikx} F(k) dk\,. \tag{11.1.1b}$$

On a practical level, the value of the Fourier transform revolves squarely around the derivative relationship

$$\widehat{f^{(n)}} = (ik)^n \widehat{f} \tag{11.1.2}$$

which results from the definition of the Fourier transform and integration by parts. Recall that we denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$.

When considering a computational domain, the solution can only be found on a finite length domain. Thus, the definition of the Fourier transform needs to be modified in order to account for the finite-sized computational domain. Instead of expanding in terms of a continuous integral for values of wavenumber k and cosines and sines ($\exp(ikx)$), we expand in a Fourier series

$$F(k) = \sum_{n=1}^{N} f(n) \exp\left[-i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le k \le N \tag{11.1.3a}$$

$$f(n) = \frac{1}{N} \sum_{k=1}^{N} F(k) \exp\left[i\frac{2\pi(k-1)}{N}(n-1)\right] \quad 1 \le n \le N. \tag{11.1.3b}$$

Thus, the Fourier transform is nothing but an expansion in a basis of cosine and sine functions. If we define the fundamental oscillatory piece as

$$w^{nk} = \exp\left(\frac{2i\pi(k-1)(n-1)}{N}\right) = \cos\left(\frac{2\pi(k-1)(n-1)}{N}\right) + i\sin\left(\frac{2\pi(k-1)(n-1)}{N}\right), \tag{11.1.4}$$

then the Fourier transform results in the expression

$$F_n = \sum_{k=0}^{N-1} w^{nk} f_k, \qquad 0 \le n \le N-1. \tag{11.1.5}$$

Thus, the calculation of the Fourier transform involves a double sum and an $O(N^2)$ operation. Thus, at first, it would appear that the Fourier transform method is the same operation count as LU decomposition. The basis functions used for the Fourier transform, sine transform, and cosine transform are depicted in Fig. 11.1. The process of solving a differential or partial differential equation involves evaluating the coefficient of each of the modes. Note that this expansion, unlike the finite difference method, is a *global* expansion in that every basis function is evaluated on the entire domain.

The Fourier, sine and cosine transforms behave very differently at the boundaries. Specifically, the Fourier transform assumes periodic boundary conditions, whereas the sine and cosine transforms assume pinned and no-flux boundaries, respectively. The cosine and sine transform are often chosen for their boundary properties. Thus, for a given problem, an evaluation must be made of the type of transform to be used based upon the boundary conditions needing to be satisfied. Table 11.1 illustrates the three different expansions and their associated boundary conditions. The appropriate Python command is also given.

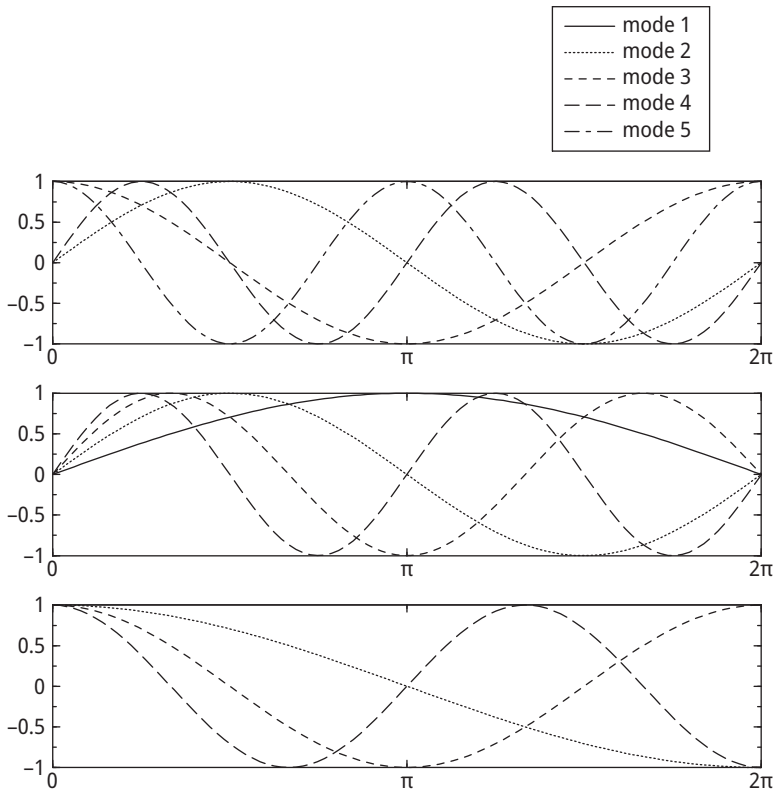


Figure 11.1 Basis functions used for a Fourier mode expansion (top), a sine expansion (middle), and a cosine expansion (bottom).

Table 11.1 Python functions for Fourier, sine, and cosine transforms and their associated boundary conditions. To invert the expansions, the Python commands are *ifft, idst*, and *idct*, respectively.

command	expansion	boundary conditions
fft	$F_k = \sum_{j=0}^{2N-1} f_j \exp(i\pi jk/N)$	periodic: $f(0) = f(L)$
dst	$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N)$	pinned: $f(0) = f(L) = 0$
dct	$F_k = \sum_{j=0}^{N-2} f_j \cos(\pi jk/2N)$	no-flux: $f'(0) = f'(L) = 0$

11.1.1 Fast Fourier transforms: Cooley–Tukey algorithm

To see how the FFT gets around the computational restriction of an $O(N^2)$ scheme, we consider the Cooley–Tukey algorithm, which drops the computations to $O(N \log N)$. To consider the algorithm, we begin with the $N \times N$ matrix $\mathbf{F}_N$ whose components are given by

$$(F_N)_{jk} = w_n^{jk} = \exp(i2\pi jk/N). \tag{11.1.6}$$

The coefficients of the matrix are points on the unit circle since $|w_n^{jk}| = 1$. They are also the basis functions for the Fourier transform.

The FFT starts with the following trivial observation:

$$w_{2n}^2 = w_n, \tag{11.1.7}$$

which is easy to show since $w_n = \exp(i2\pi/n)$ and

$$w_{2n}^2 = \exp(i2\pi/(2n)) \exp(i2\pi/(2n)) = \exp(i2\pi/n) = w_n. \tag{11.1.8}$$

The consequences of this simple relationship are enormous and are at the core of the success of the FFT algorithm. Essentially, the FFT is a matrix operation

$$\mathbf{y} = \mathbf{F}_N \mathbf{x} \tag{11.1.9}$$

which can now be split into two separate operations. Thus, defining

$$\mathbf{x}^e = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ \vdots \\ x_{N-2} \end{pmatrix} \quad \text{and} \quad \mathbf{x}^o = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{11.1.10}$$

which are both vectors of length $M = N/2$, we can form the two $M \times M$ systems

$$\mathbf{y}^e = \mathbf{F}_M \mathbf{x}^e \quad \text{and} \quad \mathbf{y}^o = \mathbf{F}_M \mathbf{x}^o \tag{11.1.11}$$

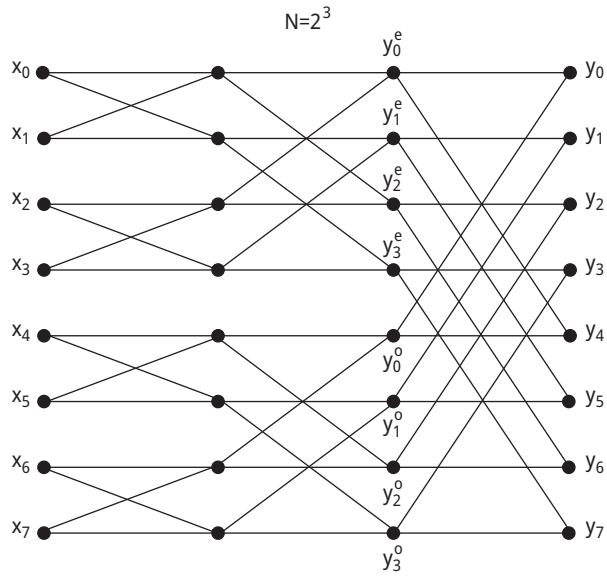


Figure 11.2 Graphical description of the fast Fourier transform process, which systematically continues to factor the problem in two. This process allows the FFT routine to drop to $O(N \log N)$ operations.

for the even coefficient terms $\mathbf{x}^e, \mathbf{y}^e$ and odd coefficient terms $\mathbf{x}^o, \mathbf{y}^o$. Thus, the computation size goes from $O(N^2)$ to $O(2M^2) = O(N^2/2)$. However, we must be able to reconstruct the original $\mathbf{y}$ from the smaller system of $\mathbf{y}^e$ and $\mathbf{y}^o$. And indeed, we can reconstruct the original $\mathbf{y}$. In particular, it can be shown that component by component

$$y_n = y_n^e + w_N^n y_n^o \quad n = 0, 1, 2, \ldots, M-1 \tag{11.1.12a}$$

$$y_{n+M} = y_n^e - w_N^n y_n^o \quad n = 0, 1, 2, \ldots, M-1\,. \tag{11.1.12b}$$

This is where the shift occurs in the FFT routine, which maps the domain $x \in [0, L]$ to $[-L, 0]$ and $x \in [-L, 0]$ to $[0, L]$. The command *fftshift* undoes this shift. Details of this construction can be found elsewhere [40; 44].

There is no reason to stop the splitting process at this point. In fact, provided we choose the size of our domain and matrix $\mathbf{F}_N$ so that N is a power of 2, we can continue to split the system until we have a simple algebraic, i.e. a 1×1 system, solution to perform. The process is illustrated graphically in Fig. 11.2, where the switching and factorization are illustrated. Once the final level is reached, the algebraic expression is solved and the process is reversed. This factorization process renders the FFT scheme $O(N \log N)$.

11.2 Chebyshev Polynomials and Transform

The fast Fourier transform is only one of many possible expansion bases, i.e. there is nothing special about expanding in cosines and sines. Of course, the FFT expansion does have the unusual property of factorization, which drops it to an $O(N \log N)$ scheme. Regardless,

there are a myriad of other expansion bases that can be considered. The primary motivation for considering other expansions is based upon the specifics of the given governing equations and its physical boundaries and constraints. Special functions are often prime candidates for use as expansion bases. The following are some important examples:

- Bessel functions: radial, 2D problems
- Legendre polynomials: 3D Laplace equation
- Hermite–Gauss polynomials: Schrödinger with harmonic potential
- Spherical harmonics: radial, 3D problems
- Chebyshev polynomials: bounded 1D domains.

The two key properties required to use any expansion basis successfully are its orthogonality properties and the calculation of the norm. Regardless, all the above expansions appear to require $O(N^2)$ calculations.

11.2.1 Chebyshev polynomials

Although not often encountered in mathematics courses, the Chebyshev polynomial is an important special function from a computational viewpoint. The reason for its prominence in the computational setting will become apparent momentarily. For the present, we note that the Chebyshev polynomials are solutions to the differential equation

$$\sqrt{1-x^2}\frac{d}{dx}\left(\sqrt{1-x^2}\frac{dT_n}{dx}\right)+n^2T_n=0 \quad x\in[-1,1]. \tag{11.2.1}$$

This is a self-adjoint Sturm–Lioville problem. Thus, the following properties are known:

(1) Eigenvalues are real: $\lambda_n = n^2$

(2) Eigenfunctions are real: $T_n(x)$

(3) Eigenfunctions are orthogonal:

$$\int_{-1}^{1}(1-x^2)^{-1/2}T_n(x)T_m(x)dx=\frac{\pi}{2}c_n\delta_{nm} \tag{11.2.2}$$

where $c_0 = 2, c_n = 1(n > 0)$, and δ_{nm} is the delta function

(4) Eigenfunctions form a complete basis.

Each Chebyshev polynomial (of degree n) is defined by

$$T_n(\cos\theta)=\cos n\theta. \tag{11.2.3}$$

Thus, we find

$$T_0(x) = 1 \tag{11.2.4a}$$

$$T_1(x) = x \tag{11.2.4b}$$

$$T_2(x) = 2x^2 - 1 \tag{11.2.4c}$$

$$T_3(x) = 4x^3 - 3x \tag{11.2.4d}$$

$$T_4(x) = 8x^4 - 8x^2 + 1. \tag{11.2.4e}$$

The behaviour of the first five Chebyshev polynomials is illustrated in Fig. 11.3.

It is appropriate to ask why the Chebyshev polynomials, of all the special functions listed, are of such computational interest. Especially given that the equation, which the $T_n(x)$ satisfy, and their functional form shown in Fig. 11.3, appear to be no better than Bessel, Hermite–Gauss, or any other special function. The distinction with the Chebyshev polynomials is that you can transform them so that use can be made of the $O(N \log N)$ discrete cosine transform. This effectively renders the Chebyshev expansion scheme an $O(N \log N)$ transformation. Specifically, we transform from the interval $x \in [-1, 1]$ by letting

$$x = \cos\theta \qquad \theta \in [0, \pi]. \tag{11.2.5}$$

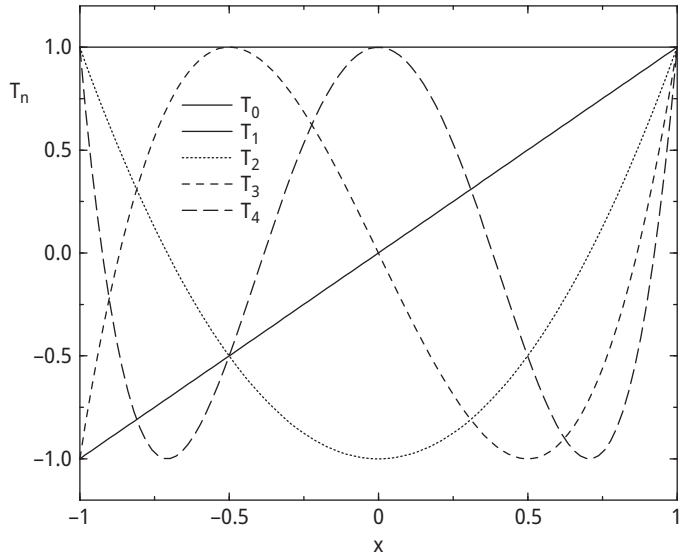


Figure 11.3 The first five Chebyshev polynomials over the interval of definition $x \in [-1, 1]$.

Thus, when considering a function $f(x)$, we have $f(\cos\theta) = g(\theta)$. Under differentiation, we find

$$\frac{dg}{d\theta} = -f' \cdot \sin\theta. \tag{11.2.6}$$

Thus, $dg/d\theta = 0$ at $\theta = 0, \pi$, i.e. no-flux boundary conditions are satisfied. This allows us to use the discrete cosine transform (*dct*) to solve a given problem in the new transformed variables.

The Chebyshev expansion is thus given by

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x) \tag{11.2.7}$$

where the coefficients a_k are determined from orthogonality and inner products to be

$$a_k = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) T_k(x) dx. \tag{11.2.8}$$

It is these coefficients which are calculated in $O(N \log N)$ time. Some of the properties of the Chebyshev polynomials are as follows:

- $T_{n+1} = 2xT_n(x) - T_{n-1}(x)$
- $|T_n(x)| \leq 1, \ |T_n'(x)| \leq n^2$
- $T_n(\pm 1) = (\pm 1)^n$
- $d^p/dx^p(T_n(\pm 1)) = (\pm 1)^{n+p} \prod_{k=0}^{p-1}(n^2 - k^2)/(2k+1)$
- If n is even (odd), $T_n(x)$ is even (odd).

There are a couple of critical practical issues, which must be considered when using the Chebyshev scheme. Specifically, the grid generation and spatial resolution are a little more difficult to handle. In using the discrete cosine transform on the variable $\theta \in [0, \pi]$, we recall that our original variable is actually $x = \cos\theta$, where $x \in [-1, 1]$. Thus, the discretization of the θ variable leads to

$$x_m = \cos\left(\frac{(2m-1)\pi}{2n}\right) \quad m = 1, 2, \ldots, n. \tag{11.2.9}$$

Thus, although the grid points are uniformly spaced in θ, the grid points are clustered in the original x variable. Specifically, there is a clustering of grid points at the boundaries. The Chebyshev scheme then automatically has higher resolution at the boundaries of the computational domain. The clustering of the grid points at the boundary is illustrated in Fig. 11.4. So, as the resolution is increased, it is important to be aware that the resolution increase is not uniform across the computational domain.

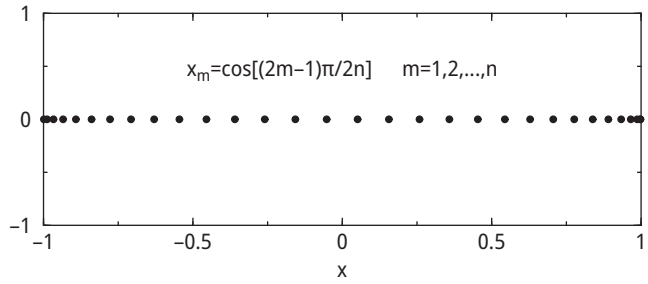


Figure 11.4 Clustered grid generation for $n = 30$ points using the Chebyshev polynomials. Note that although the points are uniformly spaced in θ, they are clustered due to the fact that $x_m = \cos[(2m-1)\pi/2n]$, where $m = 1, 2, ..., n$.

11.2.2 Solving differential equations

As with any other solution method, a solution scheme must have an efficient way of relating derivatives to the function itself. For the FFT method, there was a very convenient relationship between the transform of a function and the transform of its derivatives. Although not as transparent as the FFT method, we can also relate the Chebyshev transform derivatives to the Chebyshev transform itself.

Defining L to be a linear operator so that

$$Lf(x) = \sum_{n=0}^{\infty} b_n T_n(x), \tag{11.2.10}$$

then with $f(x) = \sum_{n=0}^{\infty} a_n T_n(x)$, we find

- $Lf = f'(x) : c_n b_n = 2 \sum_{p=n+1(p+nodd)}^{\infty} p a_p$
- $Lf = xf(x) : b_n = (c_{n-1} a_{n-1} + a_{n+1})/2$
- $Lf = x^2 f(x) : b_n = (c_{n-2} a_{n-2} + (c_n + c_{n-1}) a_n + a_{n+2})/4$

where $c_0 = 2, c_n = 0(n < 0), c_n = 1(n > 0), d_n = 1(n \geq 0)$, and $d_n = 0(n < 0)$.

11.3 Spectral Method Implementation

In this section, we develop an algorithm that implements a spectral method solution technique. We begin by considering the general partial differential equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{11.3.1}$$

where L is a linear, constant coefficient operator, i.e. it can take the form $L = ad^2/dx^2 + bd/dx + c$, where a, b, and c are constants. The second term $N(u)$ includes the nonlinear and nonconstant coefficient terms. An example of this would be $N(u) = u^3 + f(x)u + g(x)d^2u/dx^2$.

By applying a Fourier transform, the equations reduce to the system of differential equations

$$\frac{d\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{11.3.2}$$

This system can be stepped forward in time with any of the standard time-stepping techniques. Typically a Runge–Kutta scheme (4th order) is a good first attempt.

The parameter $\alpha(k)$ arises from the linear operator Lu and is easily determined from Fourier transforming. Specifically, if we consider the linear operator

$$Lu = a\frac{d^2u}{dx^2} + b\frac{du}{dx} + cu \tag{11.3.3}$$

then upon transforming, this becomes

$$\begin{aligned}(ik)^2a\widehat{u} + b(ik)\widehat{u} + c\widehat{u} \\ &= (-k^2a + ibk + c)\widehat{u} \\ &= \alpha(k)\widehat{u}.\end{aligned} \tag{11.3.4}$$

The parameter $\alpha(k)$ therefore takes into account all constant coefficient, linear differentiation terms.

The nonlinear terms are a bit more difficult to handle, but they are still relatively easy. Consider the following examples:

(1) $f(x)du/dx$
- determine $du/dx \to \widehat{du/dx} = ik\widehat{u}, du/dx = FFT^{-1}(ik\widehat{u})$
- multiply by $f(x) \to f(x)du/dx$
- Fourier transform $FFT(\,f(x)du/dx)$.

(2) u^3
- Fourier transform $FFT(u^3)$.

(3) u^3d^2u/dx^2
- determine $d^2u/dx^2 \to \widehat{d^2u/dx^2} = (ik)^2\widehat{u}, d^2u/dx^2 = FFT^{-1}(-k^2\widehat{u})$
- multiply by $u^3 \to u^3d^2u/dx^2$
- Fourier transform $FFT(u^3d^2u/dx^2)$.

These examples give an outline of how the nonlinear, nonconstant coefficient schemes would work in practice.

To illustrate the implementation of these ideas, we solve the advection–diffusion equations spectrally. Thus far, the equations have been considered largely with finite difference techniques. However, the Python codes presented here solve the equations using the FFT for both the streamfunction and vorticity evolution. We begin by initializing the appropriate numerical parameters

```
tspan = np.arange(0, 12, 2)
nu = 0.001
Lx, Ly = 20, 20
nx, ny = 64, 64
N = nx * ny

# Define spatial domain and initial conditions
x2 = np.linspace(-Lx/2, Lx/2, nx + 1)
x = x2[:nx]
y2 = np.linspace(-Ly/2, Ly/2, ny + 1)
y = y2[:ny]
```

Thus, the computational domain is $x \in [-10, 10]$ and $y \in [-10, 10]$. The diffusion parameter is chosen to be $\nu = 0.001$. The initial conditions are then defined as a stretched Gaussian

```
# initial conditions
X, Y = np.meshgrid(x, y)
w = 1 * np.exp(-0.25 * X**2 - Y**2) + 1j * np.zeros((nx, ny))
```

The next step is to define the spectral k values in both the x- and y-directions. This allows us to solve for the streamfunction (8) spectrally. Once the streamfunction is determined, the vorticity can be stepped forward in time.

```
# Define spectral k values
kx = (2 * np.pi / Lx) * np.concatenate((np.arange(0, nx/2),
np.arange(-nx/2, 0)))
kx[0] = 1e-6
ky = (2 * np.pi / Ly) * np.concatenate((np.arange(0, ny/2),
   np.arange(-ny/2, 0)))
ky[0] = 1e-6
KX, KY = np.meshgrid(kx, ky)
K = KX**2 + KY**2

# Solve the ODE and plot the results
wt0= np.hstack([np.real(fft2(w).reshape(N)),np.imag(fft2(w)
   .reshape(N))])
wtsol = odeint(spc_rhs, wt0, tspan, args=(nx, ny, N, KX, KY, K, nu))
```

The right-hand side of the system of differential equations, which results from Fourier transforming is contained within the function *spc_rhs.m*. The outline of this routine is provided below. Note that the matrix components are reshaped into a vector and a large system of differential equations is solved.

```
# Define the ODE system
def spc_rhs(wt2, t, nx, ny, N, KX, KY, K, nu):
    wtc = wt2[0:N] + 1j*wt2[N:]
    wt = wtc.reshape((nx, ny))
    psit = -wt / K
    psix = np.real(ifft2(1j * KX * psit))
    psiy = np.real(ifft2(1j * KY * psit))
    wx = np.real(ifft2(1j * KX * wt))
    wy = np.real(ifft2(1j * KY * wt))
    rhs = (-nu * K * wt + fft2(wx * psiy - wy * psix)) .reshape(N)
    return np.hstack([np.real(rhs),np.imag(rhs)])
```

The code will quickly and efficiently solve the advection–diffusion equations in two dimensions. Figure 3 demonstrates the evolution of the initial stretched Gaussian vortex over $t \in [0, 8]$.

11.4 Pseudo-Spectral Techniques with Filtering

The decomposition of the solution into Fourier mode components does not always lead to high-performance computing. Specifically, when some form of numerical stiffness is present, computational performance can suffer dramatically. However, there are methods available which can effectively eliminate some of the numerical stiffness by making use of analytic insight into the problem.

We consider again the example of hyper-diffusion for which the governing equations are

$$\frac{\partial u}{\partial t} = -\frac{\partial^4 u}{\partial x^4}. \tag{11.4.1}$$

In the Fourier domain, this becomes

$$\frac{d\,\widehat{u}}{dt} = -(ik)^4\,\widehat{u} = -k^4\,\widehat{u} \tag{11.4.2}$$

which has the solution

$$\widehat{u} = \widehat{u}_0 \exp(-k^4 t). \tag{11.4.3}$$

However, a time-stepping scheme would obviously solve (11.4.2) directly without making use of the analytic solution.

For $n = 128$ Fourier modes, the wavenumber k ranges in values from $k \in [-64, 64]$. Thus, the largest value of k for the hyper-diffusion equation is

$$k^4_{\text{max}} = (64)^4 = 16,777,216, \tag{11.4.4}$$

or roughly $k^4_{\text{max}} = 1.7 \times 10^7$. For $n = 1024$ Fourier modes, this is

$$k^4_{\text{max}} = 6.8 \times 10^{10}. \tag{11.4.5}$$

Thus, even if our solution is small at the high wavenumbers, this can create problems. For instance, if the solution at high wavenumbers is $O(10^{-6})$, then

$$\frac{d\,\widehat{u}}{dt} = -(10^{10})(10^{-6}) = -10^4. \tag{11.4.6}$$

Such large numbers on the right-hand side of the equations force time-stepping schemes to take much smaller time-steps in order to maintain tolerance constraints. This is a form of numerical stiffness that can be circumvented.

11.4.1 Filtered pseudo-spectral

There are a couple of ways to help get around the above-mentioned numerical stiffness. We again consider the very general partial differential equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{11.4.7}$$

where as before L is a linear, constant coefficient operator, i.e. it can take the form $L = ad^2/dx^2 + bd/dx + c$, where $a, b,$ and c are constants. The second term $N(u)$ includes the nonlinear and nonconstant coefficient terms. An example of this would be $N(u) = u^3 + f(x)u + g(x)d^2u/dx^2$.

Previously, we transformed the equation by Fourier transforming and constructing a system of differential equations. However, there is a better way to handle this general equation and remove some numerical stiffness at the same time. To introduce the technique, we consider the first-order differential equation

$$\frac{dy}{dt} + p(t)y = g(t). \tag{11.4.8}$$

We can multiply by the integrating factor $\mu(t)$ so that

$$\mu\frac{dy}{dt} + \mu p(t)y = \mu g(t). \tag{11.4.9}$$

We note from the chain rule that $(\mu y)' = \mu' y + \mu y'$, where the prime denotes differentiation with respect to t. Thus, the differential equation becomes

$$\frac{d}{dt}(\mu y) = \mu g(t), \tag{11.4.10}$$

provided $d\mu/dt = \mu p(t)$. The solution then becomes

$$y = \frac{1}{\mu}\left[\int \mu(t) g(t) dt + c\right] \quad \mu(t) = \exp\left(\int p(t) dt\right) \tag{11.4.11}$$

which is the standard integrating factor method of a first course on differential equations.

We use the key ideas of the integrating factor method to help solve the general partial differential equation and remove stiffness. Fourier transforming (11.4.7) results in the spectral system of differential equations

$$\frac{d\,\widehat{u}}{dt} = \alpha(k)\widehat{u} + \widehat{N(u)}\,. \tag{11.4.12}$$

This can be rewritten as

$$\frac{d\,\widehat{u}}{dt} - \alpha(k)\widehat{u} = \widehat{N(u)}\,. \tag{11.4.13}$$

Multiplying by $\exp(-\alpha(k)t)$ gives

$$\begin{aligned}
&\frac{d\,\widehat{u}}{dt}\exp(-\alpha(k)t) - \alpha(k)\widehat{u}\exp(-\alpha(k)t) = \exp(-\alpha(k)t)\widehat{N(u)} \\
&\frac{d}{dt}[\widehat{u}\exp(-\alpha(k)t)] = \exp(-\alpha(k)t)\widehat{N(u)}.
\end{aligned}$$

By defining $\widehat{v} = \widehat{u}\exp(-\alpha(k)t)$, the system of equations reduces to

$$\frac{d\,\widehat{v}}{dt} = \exp(-\alpha(k)t)\widehat{N(u)} \tag{11.4.14a}$$

$$\widehat{u} = \widehat{v}\exp(\alpha(k)t). \tag{11.4.14b}$$

Thus, the linear, constant coefficient terms are solved for explicitly, and the numerical stiffness associated with the Lu term is effectively eliminated.

11.4.2 Example: Fisher–Kolmogorov equation

As an example of the implementation of the filtered pseudo-spectral scheme, we consider the Fisher–Kolmogorov equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + u^3 + cu. \tag{11.4.15}$$

Fourier transforming the equation yields

$$\frac{d\widehat{u}}{dt} = (ik)^2\widehat{u} + \widehat{u^3} + c\widehat{u} \tag{11.4.16}$$

which can be rewritten as

$$\frac{d\widehat{u}}{dt} + (k^2 - c)\widehat{u} = \widehat{u^3}. \tag{11.4.17}$$

Thus, $\alpha(k) = c - k^2$ and

$$\frac{d\widehat{v}}{dt} = \exp[-(c - k^2)t]\widehat{u^3} \tag{11.4.18a}$$

$$\widehat{u} = \widehat{v}\exp[(c - k^2)t]. \tag{11.4.18b}$$

It should be noted that the solutions u must continually be updating the value of u^3, which is being transformed in the right-hand side of the equations. Thus, after every time-step Δt, the new u should be used to evaluate $\widehat{u^3}$.

There are a few practical issues that should be considered when implementing this technique:

- When solving the general equation

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{11.4.19}$$

  it is important to only step forward Δt in time with

$$\frac{d\widehat{v}}{dt} = \exp(-\alpha(k)t)\widehat{N(u)}$$
$$\widehat{u} = \widehat{v}\exp(\alpha(k)t)$$

  before the nonlinear term $\widehat{N(u)}$ is updated.
- The computational saving for this method generally does not manifest itself unless there are more than two spatial derivatives in the highest derivative of Lu.
- Care must be taken in handling your time-step Δt in Python since it uses adaptive time-stepping.

11.4.3 Comparison of Spectral and Finite Difference Methods

Before closing the discussion on the spectral method, we investigate the advantages and disadvantages associated with the spectral and finite difference schemes. Of particular interest are the issues of accuracy, implementation, computational efficiency and boundary conditions. The strengths and weaknesses of the schemes will be discussed.

A.: **Accuracy**

- **Finite Differences**: Accuracy is determined by the Δx and Δy chosen in the discretization. Accuracy is fairly easy to compute and generally much worse than spectral methods.
- **Spectral Method**: Spectral methods rely on a global expansion and are often called *spectrally accurate*. In particular, spectral methods have *infinite order accuracy*. Although the details of what this means will not be discussed here, it will suffice to say that they are generally of much higher accuracy than finite differences.

B.: **Implementation**

- **Finite Differences**: The greatest difficulty in implementing the finite difference schemes is generating the correct sparse matrices. Many of these matrices are very complicated with higher-order schemes and in higher dimensions. Further, when solving the resulting system $\mathbf{Ax} = \mathbf{b}$, it should always be checked whether $\det \mathbf{A} = 0$. The Python command *cond(A)* checks the condition number of the matrix. If *cond(A)*$> 10^{15}$, then $\det \mathbf{A} \approx 0$ and steps must be taken in order to solve the problem correctly.
- **Spectral Method**: The difficulty with using FFTs is the continual switching between the time or space domain and the spectral domain. Thus, it is imperative to know exactly when and where in the algorithm this switching must take place.

C.: **Computational Efficiency**

- **Finite Differences**: The computational time for finite differences is determined by the size of the matrices and vectors in solving $\mathbf{Ax} = \mathbf{b}$. Generally speaking, you can guarantee $O(N^2)$ efficiency by using LU decomposition. At times, iterative schemes can lower the operation count, but there are no guarantees about this.
- **Spectral Method**: The FFT algorithm is an $O(N \log N)$ operation. Thus, it is almost always guaranteed to be faster than the finite difference solution method, which is $O(N^2)$. Recall that this efficiency improvement comes with an increased accuracy as well. Thus making the spectral highly advantageous when implemented.

D.: **Boundary Conditions**

- **Finite Differences**: Of the above categories, spectral methods are generally better in every regard. However, finite differences are clearly superior when considering boundary conditions. Implementing the generic boundary conditions

$$\alpha u(L) + \beta \frac{du(L)}{dx} = \gamma \tag{11.4.20}$$

  is easily done in the finite difference framework. Also, more complicated computational domains may be considered. Generally, any computational domain that can be constructed of rectangles is easily handled by finite difference methods.
- **Spectral Method**: Boundary conditions are the critical limitation on using the FFT method. Specifically, only periodic boundary conditions can be considered. The use of

the discrete sine or cosine transform allows for the consideration of pinned or no-flux boundary conditions, but only odd or even solutions are admitted, respectively.

11.5 Boundary Conditions and the Chebyshev Transform

Thus far, we have focused on the use of the FFT as the primary tool for spectral methods and their implementation. However, many problems do not, in fact, have periodic boundary conditions and the accuracy and speed of the FFT is rendered useless. The Chebyshev polynomials are a set of mathematical functions that still allow for the construction of a spectral method that is both fast and accurate. The underlying concept is that Chebyshev polynomials can be related to sines and cosines, and therefore they can be connected to the FFT routine.

Before constructing the details of the Chebyshev method, we begin by considering three methods for handling nonperiodic boundary conditions.

11.5.1 Method 1: Periodic extension with FFTs

Since the FFT only handles periodic boundary conditions, we can periodically extend a general function $f(x)$ in order to make the function itself now periodic. The FFT routine can now be used. However, the periodic extension will in general generate discontinuities in the periodically extended function. The discontinuities give rise to Gibb's phenomenon: strong oscillations and errors are accumulated at the jump locations. This will greatly affect the accuracy of the scheme. So although spectral accuracy and speed is retained away from discontinuities, the errors and the jumps will begin to propagate out to the rest of the computational domain.

To see this phenomenon, Fig. 11.5(a) considers a function, which is periodically extended as shown in Fig. 11.5(b). The FFT approximation to this function is shown in Fig. 11.5(c), where the Gibb's oscillations are clearly seen at the jump locations. The oscillations clearly impact the usefulness of this periodic extension technique.

11.5.2 Method 2: Polynomial approximation with equi-spaced points

In moving away from an FFT basis expansion method, we can consider the most straightforward method available: polynomial approximation. Thus, we simply discretize the given function $f(x)$ with $N+1$ equally spaced points and fit an Nth-degree polynomial through it. This amounts to letting

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^N \tag{11.5.1}$$

where the coefficients a_n are determined by an $(N+1) \times (N+1)$ system of equations. This method easily satisfies the prescribed nonperiodic boundary conditions. Further, differentiation of such an approximation is trivial. In this case, however, Runge phenomena (polynomial

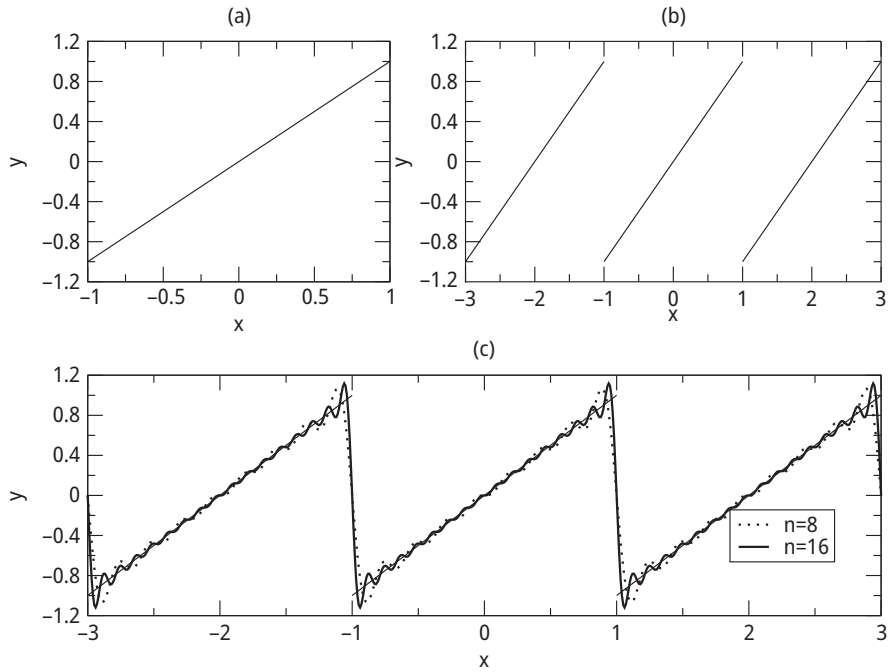


Figure 11.5 The function $y(x) = x$ for $x \in [-1, 1]$ (a) and its periodic extension (b). The FFT approximation is shown in (c) for $n = 8$ Fourier modes and $n = 16$ Fourier modes. Note the Gibb's oscillations.

oscillations) generally occur. This is because a polynomial of degree n generally has $N - 1$ combined maxima and minima.

The Runge phenomena can easily be illustrated with the simple example function $f(x) = (1 + 16x^2)^{-1}$. Figure 11.6(a)–(b) illustrates the large oscillations, which develop near the boundaries due to Runge phenomena for $N = 12$ and $N = 24$. As with the Gibb's oscillations generated by FFTs, the Runge oscillations render a simple polynomial approximation based upon equally spaced points useless.

11.5.3 Method 3: Polynomial approximation with clustered points

There exists a modification to the straightforward polynomial approximation given above. Specifically, this modification constructs a polynomial approximation on a clustered grid as opposed to the equal spacing that led to Runge phenomena. This polynomial approximation involves a clustered grid, which is transformed to fit onto the unit circle, i.e. the Chebyshev points. Thus, we have the transformation

$$x_n = \cos(n\pi/N) \tag{11.5.2}$$

where $n = 0, 1, 2, \ldots, N$. This helps to greatly reduce the effects of the Runge phenomena.

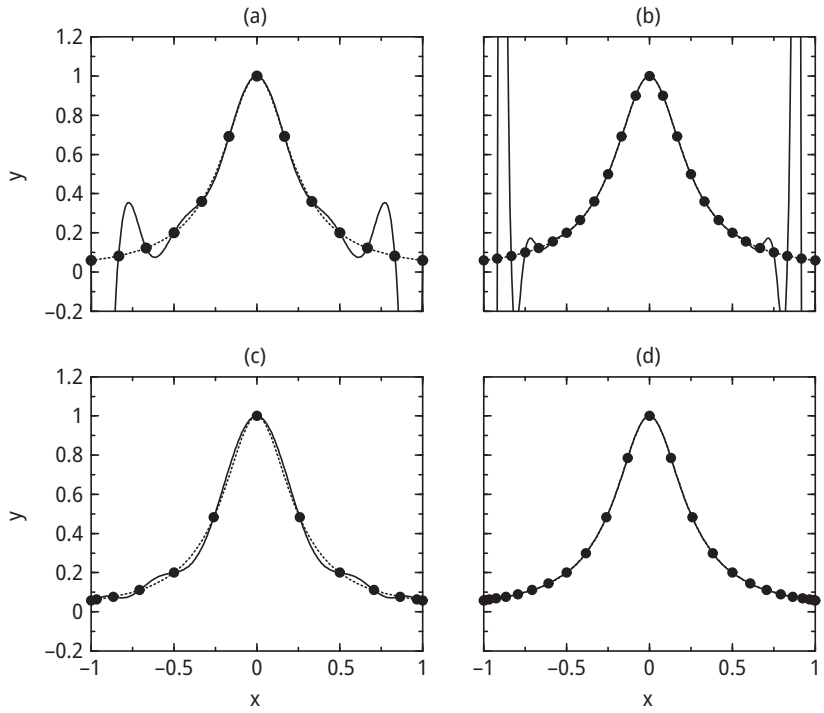


Figure 11.6 The function $y(x) = (1 + 16x^2)^{-1}$ for $x \in [-1, 1]$ (dotted line) with the bold points indicating the grid points for equi-spaced spaced points (a) $n = 12$ and (b) $n = 24$ and Chebyshev clustered points (c) $n = 12$ and (d) $n = 24$. The solid lines are the polynomial approximations generated using the equi-spaced points (a)–(b) and clustered points (c)–(d), respectively.

The clustered grid approximation results in a polynomial approximation shown in Fig. 11.6(c)–(d). There are reasons for this great improvement using a clustered grid [45]. However, we will not discuss them since they are beyond the scope of this book. This clustered grid suggests an accurate and easy way to represent a function, which does not have periodic boundaries.

11.5.4 Clustered points and Chebyshev differentiation

The clustered grid given in method 3 above is on the Chebyshev points. The resulting algorithm for constructing the polynomial approximation and differentiating it is as follows.

(1) Let p be a unique polynomial of degree $\leq N$ with $p(x_n) = V_n$, $0 \leq n \leq N$, where $V(x)$ is the function we are approximating.

(2) Differentiate by setting $w_n = p'(x_n)$.

The second step in the process of calculating the derivative is essentially the matrix multiplication

$$\mathbf{w} = \mathbf{D}_N \mathbf{v} \tag{11.5.3}$$

where $\mathbf{D}_N$ represents the action of differentiation. By using interpolation of the Lagrange form [7], the matrix elements of $p(x)$ can be constructed along with the $(N+1) \times (N+1)$ matrix $\mathbf{D}_N$. This results in each matrix element $(D_N)_{ij}$ being given by

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{11.5.4a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{11.5.4b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, \ldots, N-1 \tag{11.5.4c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, \ldots, N \ (i \neq j) \tag{11.5.4d}$$

where the parameter $c_j = 2$ for $j = 0$ or N or $c_j = 1$ otherwise.

Calculating the individual matrix elements results in the matrix $\mathbf{D}_N$

$$\mathbf{D}_N = \left(\begin{array}{c|ccc|c} \frac{2N^2+1}{6} & & \frac{2(-1)^j}{1-x_j} & & \frac{(-1)^N}{2} \\ \hline & \ddots & & \frac{(-1)^{i+j}}{x_i-x_j} & \\ \frac{-(-1)^i}{2(1-x_i)} & & \frac{-x_j}{2(1-x_j^2)} & & \frac{(-1)^{N+i}}{2(1+x_j)} \\ & \frac{(-1)^{i+j}}{x_i-x_j} & & \ddots & \\ \hline \frac{-(-1)^N}{2} & & \frac{-2(-1)^{N+j}}{1+x_j} & & -\frac{2N^2+1}{6} \end{array}\right) \tag{11.5.5}$$

which is a full matrix, i.e. it is not sparse. To calculate second, third, fourth, and higher derivatives, simply raise the matrix to the appropriate power:

- $\mathbf{D}_N^2$ – second derivative
- $\mathbf{D}_N^3$ – third derivative
- $\mathbf{D}_N^4$ – fourth derivative
- $\mathbf{D}_N^m$ – mth derivative.

11.5.5 Boundaries

The construction of the differentiation matrix $\mathbf{D}_N$ does not explicitly include the boundary conditions. Thus, the general differentiation given by (11.5.3) must be modified to include the given boundary conditions. Consider, for example, the simple boundary conditions

$$v(-1) = v(1) = 0. \tag{11.5.6}$$

The given differentiation matrix is then written as

$$\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \\ w_N \end{pmatrix} = \left(\begin{array}{c|c|c} & \text{top row} & \\ \hline \text{first column} & (D_N) & \text{last column} \\ \hline & \text{bottom row} & \end{array}\right) \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \\ v_N \end{pmatrix}. \tag{11.5.7}$$

In order to satisfy the boundary conditions, we must manually set $v_0 = v_N = 0$. Thus, only the interior points in the differentiation matrix are relevant. Note that for more general boundary conditions $v(-1) = \alpha$ and $v(1) = \beta$, we would simply set $v_0 = \alpha$ and $v_N = \beta$. The remaining $(N-1) \times (N-1)$ system is given by

$$\tilde{\mathbf{w}} = \tilde{\mathbf{D}}_N \tilde{\mathbf{v}} \tag{11.5.8}$$

where $\tilde{\mathbf{w}} = (w_1 w_2 \cdots w_{N-1})^T$, $\tilde{\mathbf{v}} = (v_1 v_2 \cdots v_{N-1})^T$, and we construct the matrix $\tilde{\mathbf{D}}_N$ with the simple Python command:

```
tildeD=D[1:(N-1),1:(N-1)]
```

Note that the new matrix created is the old $\mathbf{D}_N$ matrix with the top and bottom rows and the first and last columns removed.

11.5.6 Connecting to the FFT

We have already discussed the connection of the Chebyshev polynomial with the FFT algorithm. Thus, we can connect the differentiation matrix with the FFT routine. After transforming via (11.5.2), then for real data, the discrete Fourier transform can be used. For complex data, the regular FFT is used. Note that for the Chebyshev polynomials

$$\frac{\partial T_n(\pm 1)}{\partial x} = 0 \tag{11.5.9}$$

so that no-flux boundaries are already satisfied. To impose pinned boundary conditions $v(\pm 1) = 0$, then the differentiation matrix must be imposed as shown above.

11.6 Implementing the Chebyshev Transform

In order to make use of the Chebyshev polynomials, we must generate our given function on the clustered grid given by

$$x_j = \cos(j\pi/N) \quad j = 0, 1, 2, \ldots, N. \tag{11.6.1}$$

This clustering will give higher resolution near the boundaries than the interior of the computational domain. The Chebyshev differentiation matrix

$$\mathbf{D}_N \tag{11.6.2}$$

can then be constructed. Recall that the elements of this matrix are given by

$$(D_N)_{00} = \frac{2N^2+1}{6} \tag{11.6.3a}$$

$$(D_N)_{NN} = -\frac{2N^2+1}{6} \tag{11.6.3b}$$

$$(D_N)_{jj} = -\frac{x_j}{2(1-x_j^2)} \quad j = 1, 2, \ldots, N-1 \tag{11.6.3c}$$

$$(D_N)_{ij} = \frac{c_i(-1)^{i+j}}{c_j(x_i - x_j)} \quad i, j = 0, 1, \ldots, N \ (i \neq j) \tag{11.6.3d}$$

where the parameter $c_j = 2$ for $j = 0$ or N or $c_j = 1$ otherwise. Thus, given the number of discretization points N, we can build the matrix $\mathbf{D}_N$ and also the associated clustered grid. The following Python code simply required the number of points N to generate both of these fundamental quantities. Recall that it is assumed that the computational domain has been scaled to $x \in [-1, 1]$.

```
def cheb(N):
if N==0:
        D = 0.; x = 1.
else:
n = arange(0,N+1)
x = cos(pi*n/N).reshape(N+1,1)
c = (hstack(( [2.], ones(N-1), [2.]))*(-1)**n) .reshape(N+1,1)
X = tile(x,(1,N+1))
dX = X - X.T
D = dot(c,1./c.T)/(dX+eye(N+1))
D -= diag(sum(D.T,axis=0))
return D, x.reshape(N+1)
```

To test the differentiation, we consider two functions for which we know the exact values for the derivative and second derivative. Consider then

```
x = np.arange(-1, 1.01, 0.01)
u = np.exp(x) * np.sin(5 * x)
v = sech(x)
```

The first and second derivative of each of these functions is given by

```
ux = np.exp(x) * np.sin(5 * x) + 5 * np.exp(x) * np.cos(5 * x)
uxx = -24 * np.exp(x) * np.sin(5 * x) + 10 * np.exp(x) * np.cos(5 * x)
vx = -tanh(x) * sech(x)
vxx = sech(x) - 2 * (sech(x)**3)
```

We can also use the Chebyshev differentiation matrix to numerically calculate the values of the first and second derivatives. All that is required is the number of discretization points N and the routine *cheb.py*.

```
N = 20
D, x2 = cheb(N)
D2 = np.dot(D, D)  # Second derivative matrix
```

Given the differentiation matrix $\mathbf{D}_N$ and clustered grid, the given function and its derivatives can be constructed numerically.

```
u2 = np.exp(x2) * np.sin(5 * x2)
v2 = sech(x2)
u2x = np.dot(D, u2)
v2x = np.dot(D, v2)
u2xx = np.dot(D2, u2)
v2xx = np.dot(D2, v2)
```

A comparison between the exact values for the differentiated functions and their approximations is shown in Fig. 11.7. As is expected, the agreement is best for higher values of N. The Python code for generating these graphical figures is given by

```
plt.plot(x, u, 'r', x2, u2, '.', x, v, 'g', x2, v2, '.')
plt.plot(x, ux, 'r', x2, u2x, '.', x, vx, 'g', x2, v2x, '.')
plt.plot(x, uxx, 'r', x2, u2xx, '.', x, vxx, 'g', x2, v2xx, '.')
```

11.6.1 Differentiation matrix in 2D

Unlike finite difference schemes which result in sparse differentiation matrices, the Chebyshev differentiation matrix is full. The construction of 2D differentiation matrices thus would

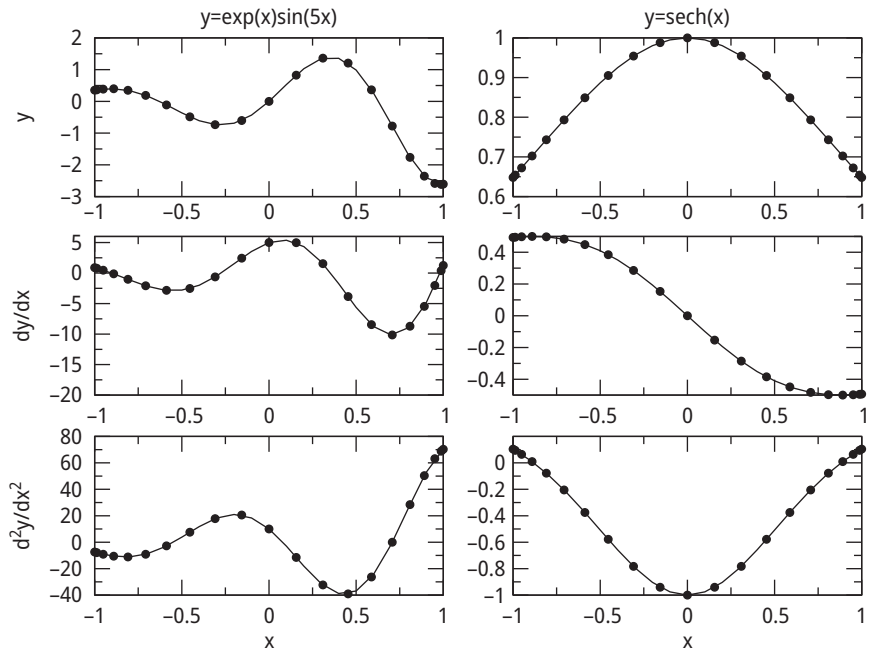


Figure 11.7 The function $y(x) = \exp(x)\sin 5x$ (left) and $y(x) = \text{sech}x$ (right) for $x \in [-1, 1]$ and their first and second derivatives. The dots indicate the numerical values, while the solid line is the exact solution. For these calculations, $N = 20$ in the differentiation matrix $\mathbf{D}_N$.

seem to be a complicated matter. However, the use of the *kron* command in Python makes this calculation trivial. In particular, the 2D Laplacian operator L given by

$$Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{11.6.4}$$

can be constructed with

```
I = np.eye(len(D2))
L = kron(I, D2) + kron(D2, I)  # 2D Laplacian
```

where $D2 = D^2$, and I is the identity matrix of size $N \times N$. The $D2$ in each slot takes the x and y derivatives, respectively.

11.6.2 Solving PDEs with the Chebyshev differentiation matrix

To illustrate the use of the Chebyshev differentiation matrix, the two-dimensional heat equation is considered:

$$\frac{\partial u}{\partial t} = \nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{11.6.5}$$

on the domain $x, y \in [-1, 1]$ with the Dirichlet boundary conditions $u = 0$.

The number of Chebyshev points is first chosen and the differentiation matrix constructed in one dimension:

```
N = 40
D, x = cheb(N)
D[N, :] = 0
D[0, :] = 0
D2 = np.dot(D, D)
```

The Neumann boundary conditions were imposed by modifying the first and last rows of the differentiation matrix D. Specifically, those rows were set to zero. The initial conditions for this simulation will be a Gaussian centred at the origin.

```
y = x
X, Y = np.meshgrid(x, y)
U = np.exp(-(X**2 + Y**2) / 0.1)
```

Note that the vectors x and y in the *meshgrid* command correspond to the Chebyshev points for a given N. The final step is to build a loop, which will advance and plot the solution in time. This requires the use of the *reshape* command, since we are in two dimensions, and an ODE solver such as *ode23*.

```
def heatrhs2D(u, t, L, mu):
    return mu*np.dot(L, u)

u = U.reshape((N + 1) ** 2)
mu = 1
for j in range(4):
    tspan = np.array([0, 0.05])
    ysol = odeint(heatrhs2D, u, tspan, args=(L,mu))
    u = ysol[-1]
    U = u.reshape(N + 1, N + 1)
```

This code will advance the solution $\Delta t = 0.05$ in the future and plot the solution. Figure 11.8 depicts the two-dimensional diffusive behaviour given the Dirichlet boundary conditions using the Chebyshev differentiation matrix.

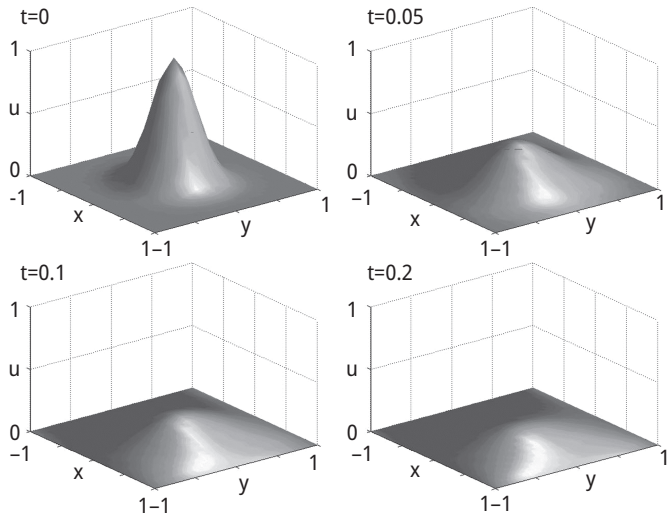


Figure 11.8 Evolution of an initial two-dimensional Gaussian governed by the heat equation on the domain $x, y \in [-1, 1]$ with Dirichlet boundary conditions $u = 0$. The Chebyshev differentiation matrix is used to calculate the Laplacian with $N = 30$.

11.7 Computing Spectra: The Floquet–Fourier–Hill Method

Spectral transforms, such as Fourier and Chebyshev, have many advantages including their performance speed of $O(N \log N)$ and their spectral accuracy properties. Given these advantages, it is desirable to see where else such methods can play a role in practice. One area that we now return to is the idea of computing spectra of linear operators. This was introduced via finite difference discretization in Section 8.8. But just as finite difference discretization of PDEs can be replaced in certain cases by spectral discretization of PDEs, so can the computation of spectra of linearized operators be replaced with spectral based methods [46].

The methodology developed here will be applied to finding spectra (all eigenvalues) of the eigenvalue problem

$$\mathcal{L}v = \lambda v \tag{11.7.1}$$

where the linear operator is assumed to be of the form

$$\mathcal{L} = \sum_{k=0}^{M} f_k(x)\partial_x^k = f_0(x) + f_1(x)\partial_x + \cdots + f_M(x)\partial_x^M \tag{11.7.2}$$

with the periodic constraint $f_k(x + L) = f_k(x)$. Thus, there are restrictions on how broadly the technique can be applied. Generically, it is for periodic problems only. However, for

problems of infinite extent where the solutions $v(\pm\infty) \to 0$, the method is often successful when considering a large, but finite-sized domain, i.e. the periodic boundary conditions suffice to approximate the decaying to zero solutions at the boundary. The Floquet–Fourier–Hill method can also be generalized to a vector version [46].

The numerical technique developed here is based upon ideas of Fourier analysis and Floquet theory. George Hill first used variations of this technique in 1886 [47] on what is now called Hill's equation. One of the great open questions of his day concerned the stability of orbits in planetary and lunar motion. As such, the method will be called the Floquet–Fourier–Hill (FFH) method [46]. The method allows for the efficient computation of the spectra of a linear operator by exploiting numerous advantages over finite difference schemes, including accuracy and speed.

The coefficients of the operator $\mathcal{L}$ in (11.7.2) are periodic with period L; thus, they can be represented by a Fourier series expansion

$$f_k(x) = \sum_{j=-\infty}^{\infty} \hat{f}_{k,j} \exp(i2\pi jx/L), \qquad k = 0, \cdots, M \tag{11.7.3}$$

where the Fourier coefficients are determined from the inner product integral

$$\hat{f}_{k,j} = \frac{1}{L} \int_{-L/2}^{L/2} f_k(x) \exp(-i2\pi jx/L) dx, \qquad k = 0, \cdots, M. \tag{11.7.4}$$

Recall that the variable i again represents the imaginary unit. Thus far, this is standard Fourier theory. In fact, the FFT algorithm computes such Fourier coefficients in $O(N \log N)$ time.

Floquet theory dictates that every bounded solution of the fundamental equation (11.7.1) is of the form [48; 49]

$$w(x) = \exp(i\mu x)\phi(x) \tag{11.7.5}$$

with $\phi(x + L) = \phi(x)$ for any fixed λ and $\mu \in [0, 2\pi/L)$. The factor $\exp(i\mu x)$ is referred to as a Floquet multiplier and $i\mu$ is the Floquet exponent [48; 49]. In different areas of science, Floquet theory may be known as monodromy theory (Hamiltonian systems, etc.) or Bloch theory (solid state theory, etc). In the context of Bloch theory, the Floquet exponent is often referred to as the quasi-momentum.

Since the function $\phi(x)$ is also periodic with a period L, it can also be expanded in a Fourier series so that

$$w(x) = \exp(i\mu x) \sum_{j=-\infty}^{\infty} \hat{\phi}_j \exp(i2\pi jx/L) = \sum_{j=-\infty}^{\infty} \hat{\phi}_j \exp(ix[\mu + 2\pi j/L]) \tag{11.7.6}$$

where

$$\hat{\phi}_j = \frac{1}{L}\int_{-L/2}^{L/2} \phi(x)\exp(-i2\pi jx/L)dx \tag{11.7.7}$$

is the jth Fourier coefficient of $\phi(x)$. Upon multiplying (11.7.1) by $\exp(-i\mu x)$ and noting that any term of the resulting equation is periodic, the nth Fourier coefficient of the eigenvalue equation becomes, after some manipulation and using orthogonality [46],

$$\sum_{m=-\infty}^{\infty}\left(\sum_{k=0}^{M}\hat{f}_{k,n-m}\left[i\left(\mu+\frac{2\pi m}{L}\right)\right]^k\right)\hat{\phi}_m = \lambda\hat{\phi}_m. \tag{11.7.8}$$

Thus, the original eigenvalue problem has been transformed into the Fourier domain and the eigenvalue problem

$$\hat{\mathcal{L}}(\mu)\hat{\phi} = \lambda\hat{\phi} \tag{11.7.9}$$

with $\hat{\phi} = (\cdots, \hat{\phi}_{-2}, \hat{\phi}_{-1}, \hat{\phi}_0, \hat{\phi}_1, \hat{\phi}_2, \cdots)^T$ and where the μ-dependence of $\hat{\mathcal{L}}$ is explicitly indicated. This is a bi-infinite matrix with the linear operator in the spectral domain defined by

$$\hat{\mathcal{L}}(\mu)_{nm} = \sum_{k=0}^{M}\hat{f}_{k,n-m}\left[i\left(\mu+\frac{2\pi m}{L}\right)\right]^k. \tag{11.7.10}$$

Note that to this point, *no approximations were used*. The problem was simply transformed to the Fourier domain. To compute the eigenvalue spectra, the bi-infinite matrix (11.7.9) is approximated by limiting the number of Fourier modes. Specifically, a cut-off wavenumber, N, is chosen that results in a matrix system of size $2N+1$:

$$\hat{\mathcal{L}}_N(\mu)\hat{\phi}_N = \lambda_N\hat{\phi}_N \tag{11.7.11}$$

where the λ_N are now numerical approximations to the true eigenvalues λ. Convergence of this scheme is considered further in Ref. [46]. Note that the **eigs** command still plays the underlying and fundamental role, but now for a transformed matrix and vector space.

11.7.1 Example: Schrödinger operators

To implement this scheme, an example is considered arising from Schrödinger operators. Specifically, the following two operators are considered:

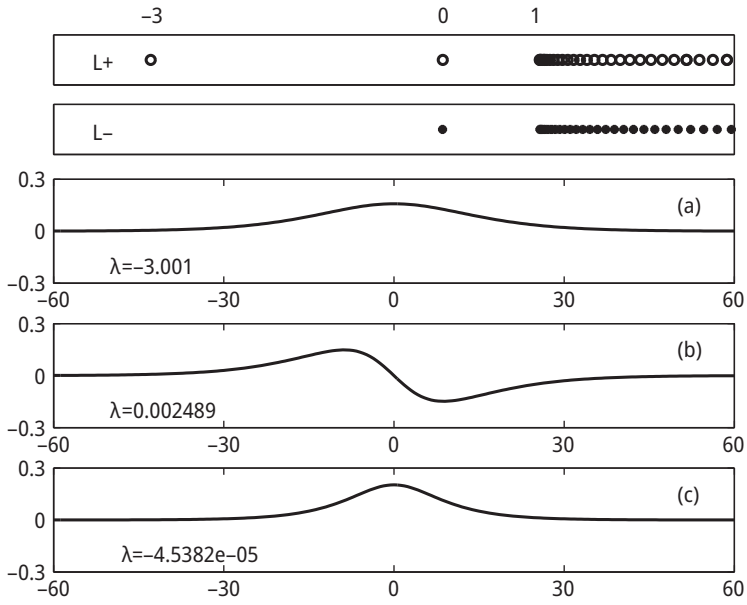


Figure 11.9 The top panels are the spectral of the operators $\mathcal{L}_+$ and $\mathcal{L}_-$. The operator $\mathcal{L}_+$ is known to have two discrete eigenvalues at $\lambda = -3$ and $\lambda = 0$ with eigenfunctions $v_{\lambda=-3} = \operatorname{sech}^2 x$ and $v_{\lambda=0} = \operatorname{sech} x \tanh x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. The panels (a) and (b) demonstrate these two eigenfunctions computed numerically along with the approximate evaluation of the eigenvalue. The operator $\mathcal{L}_-$ has a discrete eigenvalue at zero with eigenfunction $v_{\lambda=0} = \operatorname{sech} x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. Its eigenfunction and computed eigenvalue is demonstrated in panel (c).

$$\mathcal{L}_- v = -\frac{d^2 v}{dx^2} + (1 - 2\operatorname{sech}^2 x)v = \lambda v \tag{11.7.12a}$$

$$\mathcal{L}_+ v = -\frac{d^2 v}{dx^2} + (1 - 6\operatorname{sech}^2 x)v = \lambda v. \tag{11.7.12b}$$

These operators are chosen since their spectrum is known completely. In particular, the operator $\mathcal{L}_-$ has a discrete eigenvalue at zero with eigenfunction $v_{\lambda=0} = \operatorname{sech} x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$. The operator $\mathcal{L}_+$ has two discrete eigenvalues at $\lambda = -3$ and $\lambda = 0$ with eigenfunctions $v_{\lambda=-3} = \operatorname{sech}^2 x$ and $v_{\lambda=0} = \operatorname{sech} x \tanh x$. It also has a continuum of eigenvalues for $\lambda \in [1, \infty)$.

For these two examples, we have the following relations, respectively, back to (11.7.2):

$$f_0(x) = 1 - 2\operatorname{sech}^2 x, \quad f_1(x) = 0, \quad f_2(x) = -1 \tag{11.7.13a}$$

$$f_0(x) = 1 - 6\operatorname{sech}^2 x, \quad f_1(x) = 0, \quad f_2(x) = -1. \tag{11.7.13b}$$

The only Fourier expansion that needs to be done then is for the term $\operatorname{sech}^2 x$. The following Python code constructs the FFH eigenvalue problem.

```
from scipy.integrate import quad

L = 40  # box size  -L,L
n = 200  # modes
TOL = 1e-8

a = np.zeros(n + 1)
for j in range(1, n + 2):
    def integrand(x, j, L):
        return (1 / (2 * L)) * np.cos((j - 1) * np.pi * x / L)
            * sech(x) ** 2

    a[j - 1], _ = quad(integrand, -L, L, args=(j, L),
        epsabs=TOL, epsrel=TOL)

A = np.zeros((n + 1, n + 1))
for j in range(1, n + 2):
    A[j - 1, j - 1] = a[0]

for jj in range(2, n + 2):
    for j in range(jj, n + 2):
        A[j - jj, j - 1] = a[jj - 1]
        A[j - 1, j - jj] = a[jj - 1]

D2 = np.zeros((n + 1, n + 1))
for j in range(1, n + 2):
    D2[j - 1, j - 1] = -1 - ((np.pi / L) ** 2) * (n / 2 + 1 - j) ** 2

Lplus = D2 + 6 * A
lam, V = np.linalg.eig(Lplus)
lam = np.real(lam)

sorted_indices = np.argsort(np.real(lam))[::-1]
lamsort = lam[sorted_indices]
Vsort =V[:, sorted_indices]

Lminus = D2 + 2 * A
lam2, V2 = np.linalg.eig(Lminus)
lam2 = np.real(lam2)

sorted_indices = np.argsort(np.real(lam2))[::-1]
lam2sort = lam2[sorted_indices]
V2sort =V2[:, sorted_indices]
```

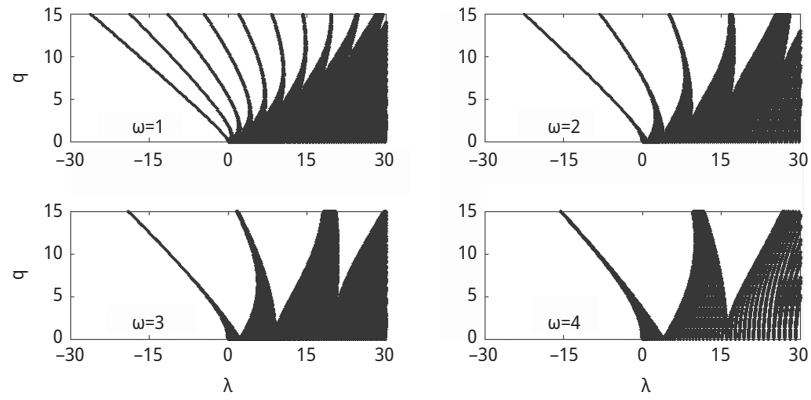


Figure 11.10 Spectrum of Mathieu's equation for four different values of ω. Every black point is an approximate point of the spectrum and no filling in of spectral regions was done. Standard perturbation theory is typically only able to approximate the spectrum near $q = 0$.

In this example where the operator is of the Sturm–Liouville form, the inner product only needs to be computed with respect to $\cos(n\pi x/L)$ versus $\exp(in\pi x/L)$. Figure 11.9 shows the computed spectrum of both the $\mathcal{L}_+$ and $\mathcal{L}_-$ operators. The eigenfunctions are also shown for the three discrete eigenvalues, two for $\mathcal{L}_+$ and one for $\mathcal{L}_-$, respectively. The computation uses $\Delta x = 0.6$, which would only give an accuracy of 10^{-1}. However, the computation with FFH is accurate to 10^{-3} or more. Note that using the **quad** command is a highly inefficient way to compute the Fourier coefficients. Using the FFT algorithm is much faster and ultimately the way to do the computation. However, the above does illustrate the FFH process in its entirety.

11.7.2 Example: Mathieu's equation

As a second example of computing spectra in a highly efficient way, consider the Mathieu equation of mathematical physics

$$\frac{d^2v}{dx^2} + [\lambda - 2q\cos(\omega x)]\, v = 0. \tag{11.7.14}$$

This gives the linear operator $\mathcal{L} = -d^2/dx^2 + 2q\cos(\omega x)$. The equation originates from the Helmholtz equation through the use of separation of variables. Here we are interested in determining all λ values for which bounded solutions exist. Many texts on perturbation methods use this equation as one of their prototypical examples. One of their goals is to determine the edges of the spectrum for varying q.

In this case, we have the following relations back to (11.7.2):

$$f_0(x) = 2q\cos(\omega x), \quad f_1(x) = 0, \quad f_2(x) = -1. \tag{11.7.15}$$

Thus, a fairly simple structure exists with $f_0(x)$ already being represented as Fourier modes if we choose ω to be an integer. In what follows, ω is chosen to be an integer and the eigenvalues are found as a function of the variable q.

```
q_values = np.arange(0, 15.1, 0.1)  # q values to consider
n = 10  # number of Fourier modes
cuts = 40  # number of mu slices
freq = 2  # frequency omega

for q in q_values:   # Loop over q values
    lam = []
    for jcut in np.arange(-freq / 2, freq / 2 + freq / cuts,
        freq / cuts):
        A = np.zeros((2 * n + 1, 2 * n + 1))
        for j in range(1, 2 * n + 2):
            A[j - 1, j - 1] = (n + 1 - j + jcut) ** 2
                # derivative elements

        for j in range(1, 2 * n + 2 - freq):
            A[j + freq - 1, j - 1] = q
            A[j - 1, j + freq - 1] = q

        W, V = np.linalg.eig(A)  # compute eigenvalues
        lam.extend(W.real)  # track all eigenvalues
```

Unlike the previous example, no Fourier mode projection needs to be done since $f_0(x) = 2q\cos(\omega x)$ is already in Fourier mode form. Thus, only the coefficient $2q$ is needed at frequency ω. Figure 11.10 shows the resulting band-gap structure of eigenvalues for four different values of ω. All points in the plot are computed points and no filling-in of the spectrum was done.

To see how to generalize the FFH method to a vector model and/or higher dimensions, see Ref. [46]. In general, one should think about this spectral based FFH method as simply taking advantage of spectral accuracy and speed. Just as PDE-based solvers aim to take advantage of these features for accurately and quickly solving a given problem, the FFH should always be used when either an infinite domain (with decaying boundary conditions) or periodic solutions are of interest.

12 Finite Element Methods

12.1 Finite Element Basis 298
12.1.1 Domain discretization 299
12.1.2 Interpolating functions 300
12.1.3 1D simplex 300
12.1.4 2D simplex 302
12.2 Discretizing with Finite Elements and Boundaries 303
12.2.1 The variational principle 304
12.2.2 Solution method 306

The numerical methods considered thus far, i.e. finite difference and spectral, are powerful tools for solving a wide variety of physical problems. However, neither method is well suited for complicated boundary configurations or complicated boundary conditions associated with such domains. The finite element method is ideally suited for complex boundaries and very general boundary conditions. This method, although perhaps not as fast as finite difference and spectral, is instrumental in solving a wide range of problems, which is beyond the grasp of other techniques.

12.1 Finite Element Basis

The finite difference method approximates a solution by discretizing and solving the matrix problem $\mathbf{Ax} = \mathbf{b}$. Spectral methods use the FFT to expand the solution in global sine and cosine basis functions. Finite elements are another form of finite difference in which the approximation to the solution is made by interpolating over patches of our solution region. Ultimately, to find the solution, we will once again solve a matrix problem $\mathbf{Ax} = \mathbf{b}$. Five essential steps are required in the finite element method:

- Discretization of the computational domain
- Selection of the interpolating functions
- Derivation of characteristic matrices and vectors
- Assembly of characteristic matrices and vectors
- Solution of the matrix problem $\mathbf{Ax} = \mathbf{b}$.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0013

As will be seen, each step in the finite element method is mathematically significant and relatively challenging. However, there are a large number of commercially available packages that take care of implementing the above ideas. Keep in mind that the primary reason to develop this technique is boundary conditions: both complicated domains and general boundary conditions.

12.1.1 Domain discretization

Finite element domain discretization usually involves the use of a commercial package. Two commonly used packages are the Python PDE Toolbox and FEMLAB (which is built on the Python platform). Writing your own code to generate an unstructured computational grid is a difficult task and well beyond the scope of this book. Thus, we will use commercial packages to generate the computational mesh necessary for a given problem. The key idea is to discretize the domain with triangular elements (or another appropriate element function). Figure 12.1 shows the discretization of a domain with complicated boundaries inside the computationally relevant region. Note that the triangular discretization is such that it adaptively sizes the triangles so that higher resolution is automatically in place where needed. The key features of the discretization are as follows:

- The width and height of all discretization triangles should be similar.
- All shapes used to span the computational domain should be approximated by polygons.

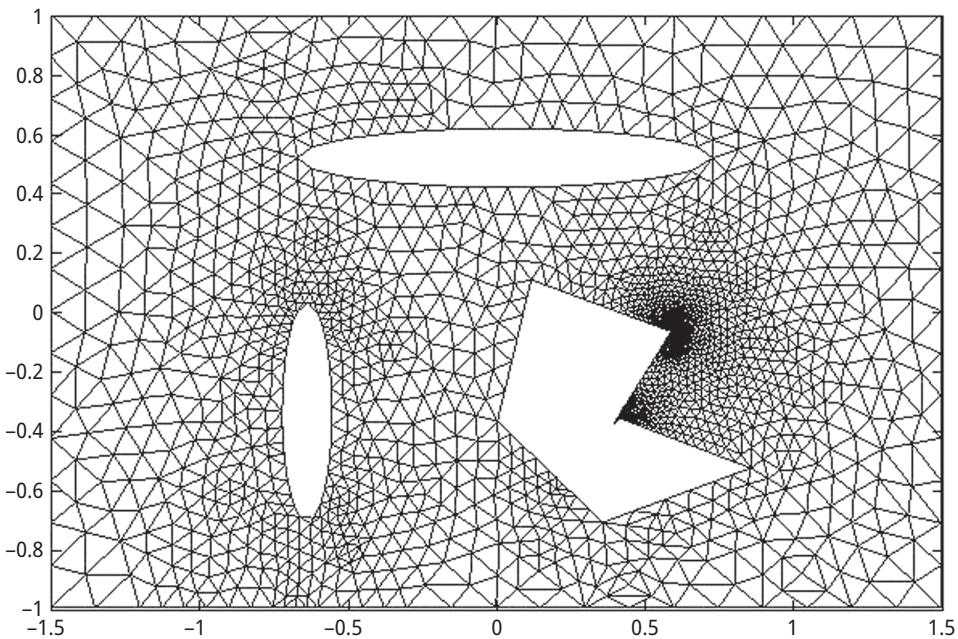


Figure 12.1 Finite element triangular discretization provided by the MATLAB PDE Toolbox. Note the increased resolution near corners and edges.

- A commercial package is almost always used to generate the grid unless you are doing research in this area.

12.1.2 Interpolating functions

Each element in the finite element basis relies on a piecewise approximation. Thus, the solution to the problem is approximated by a simple function in each triangular (or polygonal) element. As might be expected, the accuracy of the scheme depends upon the choice of the approximating function chosen.

Polynomials are the most commonly used interpolating functions. The simpler the function, the less computationally intensive it is. However, accuracy is usually increased with the use of higher-order polynomials. Three groups of elements are usually considered in the basic finite element scheme:

- Simplex elements: linear polynomials are used
- Complex elements: higher-order polynomials are used
- Multiplex elements: rectangles are used instead of triangles.

An example of each of the three finite elements is given in Figs. 12.2 and 12.3 for one-dimensional and two-dimensional finite elements.

12.1.3 1D simplex

To consider the finite element implementation, we begin by considering the one-dimensional problem. Figure 12.4 shows the linear polynomial used to approximate the solution between points x_i and x_j. The approximation of the function is thus

$$\phi(x) = a + bx = (1 \;\; x) \begin{pmatrix} a \\ b \end{pmatrix}. \tag{12.1.1}$$

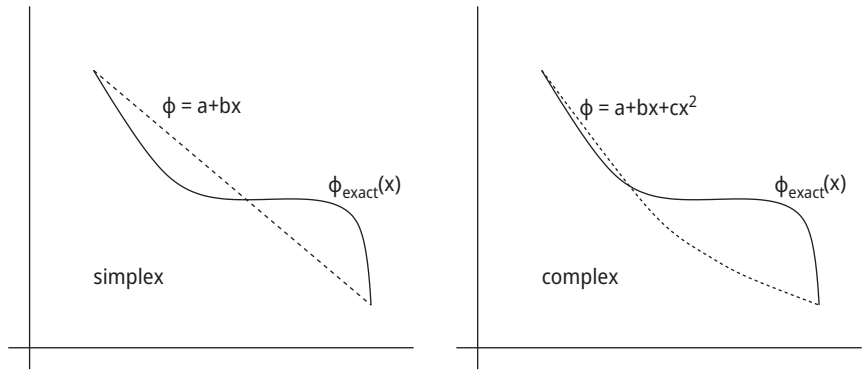


Figure 12.2 Finite element discretization for simplex and complex finite elements. Essentially the finite elements are approximating polynomials.

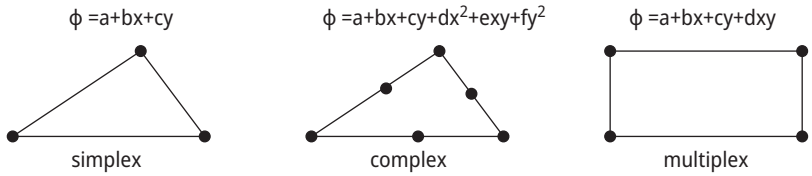


Figure 12.3 Finite element discretization for simplex, complex, and multiplex finite elements in two-dimensions. The finite elements are approximating surfaces.

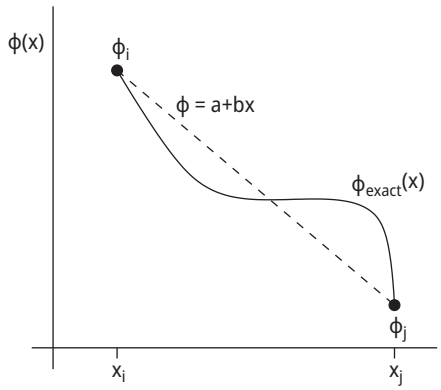


Figure 12.4 One dimensional approximation using finite elements.

The coefficients a and b are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i \tag{12.1.2a}$$

$$\phi_j = a + bx_j \tag{12.1.3b}$$

which is a 2×2 system for the unknown coefficients. In matrix form, this can be written as

$$\phi = \mathbf{A}\mathbf{a} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}, \; \mathbf{A} = \begin{pmatrix} 1 & x_i \\ 1 & x_j \end{pmatrix}, \; \mathbf{a} = \begin{pmatrix} a \\ b \end{pmatrix}. \tag{12.1.4}$$

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{x_j - x_i}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} = \frac{1}{l}\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix} \tag{12.1.5}$$

where $l = x_j - x_i$. Recalling that $\phi = (1 \;\; x)\mathbf{a}$ then gives

$$\phi = \frac{1}{l}(1 \;\; x)\begin{pmatrix} x_j & -x_i \\ -1 & 1 \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \end{pmatrix}. \tag{12.1.6}$$

Multiplying this expression out yields the approximate solution

$$\phi(x) = N_i(x)\phi_i + N_j(x)\phi_j \tag{12.1.7}$$

where $N_i(x)$ and $N_j(x)$ are the Lagrange polynomial coefficients [7] (shape functions)

$$N_i(x) = \frac{1}{l}(x_j - x) \tag{12.1.8a}$$

$$N_j(x) = \frac{1}{l}(x - x_i). \tag{12.1.8b}$$

Note the following properties: $N_i(x_i) = 1, N_i(x_j) = 0$ and $N_j(x_i) = 0, N_j(x_j) = 1$. This completes the generic construction of the polynomial, which approximates the solution in one dimension.

12.1.4 2D simplex

In two dimensions, the construction becomes a bit more difficult. However, the same approach is taken. In this case, we approximate the solution over a region with a plane (see Fig. 12.5)

$$\phi(x) = a + bx + cy. \tag{12.1.9}$$

The coefficients a, b, and c are determined by enforcing that the function goes through the end points. This gives

$$\phi_i = a + bx_i + cy_i \tag{12.1.10a}$$

$$\phi_j = a + bx_j + cy_j \tag{12.1.10b}$$

$$\phi_k = a + bx_k + cy_k \tag{12.1.10c}$$

which is now a 3×3 system for the unknown coefficients. In matrix form, this can be written

$$\phi = \mathbf{A}\mathbf{a} \;\rightarrow\; \phi = \begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix}, \mathbf{A} = \begin{pmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{pmatrix}, \mathbf{a} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}. \tag{12.1.11}$$

The geometry of this problem is reflected in Fig. 12.5, where each of the points of the discretization triangle is illustrated.

Solving for $\mathbf{a}$ gives

$$\mathbf{a} = \mathbf{A}^{-1}\phi = \frac{1}{2S}\begin{pmatrix} x_jy_k - x_ky_j & x_ky_i - x_iy_k & x_iy_j - x_jy_i \\ y_j - y_k & y_k - y_i & y_i - y_j \\ x_k - x_j & x_i - x_k & x_j - x_i \end{pmatrix}\begin{pmatrix} \phi_i \\ \phi_j \\ \phi_k \end{pmatrix} \tag{12.1.12}$$

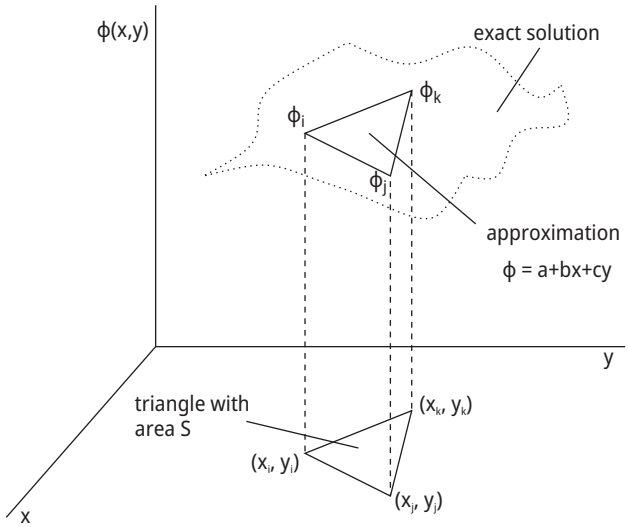


Figure 12.5 Two-dimensional triangular approximation of the solution using the finite element method.

where S is the area of the triangle projected onto the $x-y$ plane (see Fig. 12.5). Recalling that $\phi = \mathbf{Aa}$ gives

$$\phi(x) = N_i(x,y)\phi_i + N_j(x,y)\phi_j + N_k(x,y)\phi_k \tag{12.1.13}$$

where $N_i(x,y)$, $N_j(x,y)$, $N_k(x,y)$ are the Lagrange polynomial coefficients [7] (shape functions)

$$N_i(x,y) = \frac{1}{2S}\left[(x_j y_k - x_k y_j) + (y_j - y_k)x + (x_k - x_j)y\right] \tag{12.1.14a}$$

$$N_j(x,y) = \frac{1}{2S}\left[(x_k y_i - x_i y_k) + (y_k - y_i)x + (x_i - x_k)y\right] \tag{12.1.14b}$$

$$N_k(x,y) = \frac{1}{2S}\left[(x_i y_j - x_j y_i) + (y_i - y_j)x + (x_j - x_i)y\right]. \tag{12.1.14c}$$

Note the following end point properties: $N_i(x_i,y_i) = 1,\ N_j(x_j,y_j) = N_k(x_k,y_k) = 0$, and $N_j(x_j,y_j) = 1,\ N_j(x_i,y_i) = N_j(x_k,y_k) = 0$, and $N_k(x_k,y_k) = 1,\ N_k(x_i,y_i) = N_k(x_j,y_j) = 0$. This completes the generic construction of the polynomial, which approximates the solution in one dimension, and the generic construction of the surface, which approximates the solution in two dimensions.

12.2 Discretizing with Finite Elements and Boundaries

In the previous section, an outline was given for the construction of the polynomials, which approximate the solution over a finite element. Once constructed, however, they must be used to solve the physical problem of interest. The finite element solution method relies

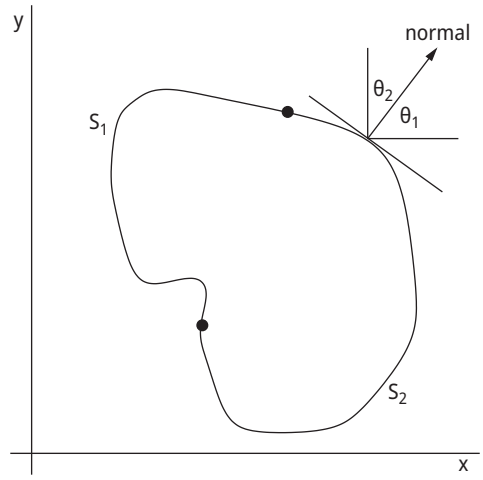


Figure 12.6 Computational domain of interest which has the two domain regions S_1 and S_2.

on solving the governing set of partial differential equations in the *weak formulation* of the problem, i.e. the integral formulation of the problem. This is because we will be using linear interpolation pieces whose derivatives do not necessarily match across elements. In the integral formulation, this does not pose a problem.

We begin by considering the elliptic partial differential equation

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right)+r(x,y)u=f(x,y) \tag{12.2.1}$$

where, over part of the boundary,

$$u(x,y)=g(x,y) \quad \text{on } S_1 \tag{12.2.2}$$

and

$$p(x,y)\frac{\partial u}{\partial x}\cos\theta_1+q(x,y)\frac{\partial u}{\partial y}\cos\theta_2+g_1(x,y)u=g_2(x,y) \quad \text{on } S_2. \tag{12.2.3}$$

The boundaries and domain are illustrated in Fig. 12.6. Note that the normal derivative determines the angles θ_1 and θ_2. A domain such as this would be very difficult to handle with finite difference techniques. And further, because of the boundary conditions, spectral methods cannot be implemented. Only the finite element method renders the problem tractable.

12.2.1 The variational principle

To formulate the problem correctly for the finite element method, the governing equations and their associated boundary conditions are recast in an integral form. This recasting of

the problem involves a variational principle. Although we will not discuss the calculus of variations here [50], the highlights of this method will be presented.

The variational method expresses the governing partial differential as an integral, which is to be minimized. In particular, the functional to be minimized is given by

$$I(\phi) = \int\int\int_V F\left(\phi, \frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) dV + \int\int_S g\left(\phi, \frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) dS, \tag{12.2.4}$$

where a three-dimensional problem is being considered. The derivation of the functions F, which captures the governing equation, and g, which captures the boundary conditions, follow the Euler–Lagrange equations for minimizing variations:

$$\frac{\delta F}{\delta\phi} = \frac{\partial}{\partial x}\left(\frac{\partial F}{\partial(\phi_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial F}{\partial(\phi_y)}\right) + \frac{\partial}{\partial z}\left(\frac{\partial F}{\partial(\phi_z)}\right) - \frac{\partial F}{\partial\phi} = 0. \tag{12.2.5}$$

This is essentially a generalization of the concept of the zero derivative, which minimizes a function. Here we are minimizing a functional.

For our governing elliptic problem (12.2.1) with boundary conditions given by those of S_2 (12.2.3), we find the functional $I(u)$ to be given by

$$\begin{aligned} I(u) = \frac{1}{2}\int\int_D dxdy\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right] \\ + \int_{S_2} dS\left[-g_2(x,y)u + \frac{1}{2}g_1(x,y)u^2\right]. \end{aligned} \tag{12.2.6}$$

Thus, the interior domain over which we integrate D has the integrand

$$I_D = \frac{1}{2}\left[p(x,y)\left(\frac{\partial u}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial u}{\partial y}\right)^2 - r(x,y)u^2 + 2f(x,y)u\right]. \tag{12.2.7}$$

This integrand was derived via variational calculus as the minimization over the functional space of interest. To confirm this, we apply the variational derivative as given by (12.2.5) and find

$$\begin{aligned} \frac{\delta I_D}{\delta u} &= \frac{\partial}{\partial x}\left(\frac{\partial I_D}{\partial(u_x)}\right) + \frac{\partial}{\partial y}\left(\frac{\partial I_D}{\partial(u_y)}\right) - \frac{\partial I_D}{\partial u} = 0 \\ &= \frac{\partial}{\partial x}\left(\frac{1}{2}p(x,y)\cdot 2\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{2}q(x,y)\cdot 2\frac{\partial u}{\partial y}\right) - \left(-\frac{1}{2}r(x,y)\cdot 2u + f(x,y)\right) \\ &= \frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u - f(x,y) = 0. \end{aligned} \tag{12.2.8}$$

Upon rearranging, this gives back the governing equation (12.2.1)

$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u = f(x,y). \tag{12.2.9}$$

A similar procedure can be followed on the boundary terms to derive the appropriate boundary conditions (12.2.2) or (12.2.3). Once the integral formulation has been achieved, the finite element method can be implemented with the following key ideas:

- The solution is expressed in the *weak* (integral) form since the linear (simplex) interpolating functions give that the second derivatives (e.g. ∂_x^2 and ∂_y^2) are zero. Thus, the elliptic operator (12.2.1) would have no contribution from a finite element of the form $\phi = a_1 + a_2 x + a_3 y$. However, in the integral formulation, the second derivative terms are proportional to $(\partial u/\partial x)^2 + (\partial u/\partial y)^2$, which gives $\phi = a_2^2 + a_3^2$.
- A solution is sought that takes the form

$$u(x,y) = \sum_{i=1}^{m} \gamma_i \phi_i(x,y) \tag{12.2.10}$$

  where $\phi_i(x,y)$ are the linearly independent piecewise-linear polynomials, and $\gamma_1, \gamma_2, \ldots, \gamma_m$ are constants.
  - $\gamma_{n+1}, \gamma_{n+2}, \ldots, \gamma_m$ ensure that the boundary conditions are satisfied, i.e. those elements which touch the boundary must meet certain restrictions.
  - $\gamma_1, \gamma_2, \ldots, \gamma_n$ ensure that the integrand $I(u)$ in the interior of the computational domain is minimized, i.e. $\partial I/\partial \gamma_i = 0$ for $i = 1, 2, 3, \ldots, n$.

12.2.2 Solution method

We begin with the governing equation in integral form (12.2.6) and assume an expansion of the form (12.2.10). This gives

$$\begin{aligned} I(u) &= I\left(\sum_{i=1}^{m} \gamma_i \phi_i(x,y)\right) \\ &= \frac{1}{2}\int\int_D dxdy \left[p(x,y)\left(\sum_{i=1}^{m}\gamma_i \frac{\partial \phi_i}{\partial x}\right)^2 + q(x,y)\left(\sum_{i=1}^{m}\gamma_i \frac{\partial \phi_i}{\partial y}\right)^2\right. \\ &\qquad \left. - r(x,y)\left(\sum_{i=1}^{m}\gamma_i \phi_i(x,y)\right)^2 + 2f(x,y)\left(\sum_{i=1}^{m}\gamma_i \phi_i(x,y)\right)\right] \\ &\quad + \int_{S_2} dS \left[-g_2(x,y)\sum_{i=1}^{m}\gamma_i \phi_i(x,y) + \frac{1}{2}g_1(x,y)\left(\sum_{i=1}^{m}\gamma_i \phi_i(x,y)\right)^2\right]. \end{aligned} \tag{12.2.11}$$

This completes the expansion portion.

The functional $I(u)$ must now be differentiated with respect to all the interior points and minimized. This involves differentiating the above with respect to γ_i, where $i = 1, 2, 3, \ldots, n$. This results in the rather complicated expression

$$\frac{\partial I}{\partial \gamma_j} = \int\int_D dxdy \left[p(x,y) \sum_{i=1}^{m} \gamma_i \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + q(x,y) \sum_{i=1}^{m} \gamma_i \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right.$$
$$\left. - r(x,y) \sum_{i=1}^{m} \gamma_i \phi_i \phi_j + f(x,y)\phi_j \right] \tag{12.2.12}$$
$$+ \int_{S_2} dS \left[-g_2(x,y)\phi_j + g_1(x,y) \sum_{i=1}^{m} \gamma_i \phi_i \phi_j \right] = 0.$$

.

Term by term, this results in an expression for the γ_i given by

$$\sum_{i=1}^{m} \left\{ \int\int_D dxdy \left[p \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + q \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} - r\phi_i \phi_j \right] + \int_{S_2} dS g_1 \phi_i \phi_j \right\} \gamma_i$$
$$+ \int\int_D dxdy f \phi_j - \int_{S_2} dS g_2(x,y)\phi_j = 0. \tag{12.2.13}$$

But this expression is simply a matrix solve, $\mathbf{Ax} = \mathbf{b}$, for the unknowns γ_i. In particular, we have

$$\mathbf{x} = \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_n \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} \quad \mathbf{A} = (\alpha_{ij}) \tag{12.2.14}$$

where

$$\beta_i = -\int\int_D dxdy f \phi_i + \int_{S_2} dS g_2 \phi_i - \sum_{k=n+1}^{m} \alpha_{ik}\gamma_k \tag{12.2.15a}$$

$$\alpha_{ij} = \int\int_D dxdy \left[p \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + q \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} - r\phi_i \phi_j \right] + \int_{S_2} dS g_1 \phi_i \phi_j. \tag{12.2.15b}$$

Recall that each element ϕ_i is given by the simplex approximation

$$\phi_i = \sum_{i=1}^{3} N_j^{(i)}(x,y)\phi_j^{(i)} = \sum_{i=1}^{3} \left(a_j^{(i)} + b_j^{(i)} x + c_j^{(i)} y \right) \phi_j^{(i)}. \tag{12.2.16}$$

This concludes the construction of the approximate solution in the finite element basis. From an algorithmic point of view, the following procedure would be carried out to generate the solution.

- Discretize the computational domain into triangles. $T_1, T_2, \ldots, T_k$ are the interior triangles, and $T_{k+1}, T_{k+2}, \ldots, T_m$ are triangles that have at least one edge that touches the boundary.

- For $l = k+1, k+2, \ldots, m$, determine the values of the vertices on the triangles that touch the boundary.
- Generate the shape functions

$$N_j^{(i)} = a_j^{(i)} + b_j^{(i)} x + c_j^{(i)} y \tag{12.2.17}$$

  where $i = 1, 2, 3, \ldots, m$ and $j = 1, 2, 3, \ldots, m$.
- Compute the integrals for matrix elements α_{ij} and vector elements β_j in the interior and boundary.
- Construct the matrix $\mathbf{A}$ and vector $\mathbf{b}$.
- Solve $\mathbf{Ax} = \mathbf{b}$.
- Plot the solution $u(x, y) = \sum_{i=1}^{m} \gamma_i \phi_i(x, y)$.

For a wide variety of problems, the above procedures can simply be automated. That is exactly what commercial packages do. Thus, once the coefficients and boundary conditions associated with (12.2.1), (12.2.2), and (12.2.3) are known, the solution procedure is straightforward.

Open source software

Although one could develop code for mesh construction and PDE solutions with finite elements (or finite volumes), there exist well-developed computing platforms offering stable and robust implementations of both. Software like GOMA, FEniCS, or openFOAM offer incredible computing packages, which allow for the simulation of complex, multiscale physics problems without having to develop the infrastructure of geometry, adaptive meshing, and differentiation. These platforms allow for research-level computing and are highly encouraged if the need for finite elements arises. As such, no homework exercises are proposed here.

PART 3

Computational Methods for Data Analysis

Statistical Methods and Their Applications

13

13.1 Basic Probability Concepts 311
13.1.1 Sample space and events 312
13.1.2 Probabilities defined on events 314
13.1.3 Conditional probabilities 315
13.1.4 Independent events 316
13.1.5 Bayes's formula 317
13.2 Random Variables and Statistical Concepts 318
13.2.1 Expectation, moments, and variance of random variables 324
13.2.2 Joint probability distribution and covariance 326
13.2.3 Python commands 327
13.3 Hypothesis Testing and Statistical Significance 327
13.3.1 Limit theorems 328
13.3.2 Statistical decisions 329
13.3.3 Tests for statistical significance 330
13.3.4 Hypothesis testing with the normal distribution 330
13.3.4.1 Example: A fair coin 331
13.3.4.2 Example: Class scores 332
13.3.4.3 Example: Rope breaking (Student's t-distribution) 333

Our ultimate goal is to analyse highly generic data arising from applications as diverse as imaging, biological sciences, atmospheric sciences, or finance, to name a few specific examples. In all these application areas, there is a fundamental reliance on extracting meaningful trends and information from large data sets. Primarily, this is motivated by the fact that in many of these systems, the degree of complexity or the governing equations is unknown or impossible to extract. Thus, one must rely on data, its statistical properties, and its analysis in the context of spectral methods and linear algebra.

13.1 Basic Probability Concepts

To understand the methods required for data analysis, a review of probability theory and statistical concepts is necessary. Many of the ideas presented in this section are intuitively understood by most students in the mathematical, biological, physical, and engineering sciences. Regardless, a review will serve to refresh one with the concepts and will further help understand their implementation in Python.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0014

13.1.1 Sample space and events

Often one performs an experiment whose outcome is not known in advance. However, while the outcome is not known, suppose that the set of all possible outcomes is known. The set of all possible outcomes is the *sample space* of the experiment and is denoted by S. Some specific examples of sample spaces are the following:

(1) If the experiment consists of flipping a coin, then

$$S = \{H, T\}$$

where H denotes an outcome of heads and T denotes an outcome of tails.

(2) If the experiment consists of flipping two coins, then

$$S = \{(H, H), (H, T), (T, H), (T, T)\}$$

where the outcome (H, H) denotes both coins being heads, (H, T) denotes the first coin being heads and the second tails, (T, H) denotes the first coin being tails and the second being heads, and (T, T) denotes both coins being tails.

(3) If the experiment consists of tossing a die, then

$$S = \{1, 2, 3, 4, 5, 6\}$$

where the outcome i means that the number i appeared on the die, $i = 1, 2, 3, 4, 5, 6$.

Any subset of the sample space is referred to as an event, and it is denoted by E. For the sample spaces given above, we can also define an event.

(1) If the experiment consists of flipping a single coin, then

$$E = \{H\}$$

denotes the event that a head appears on the coin.

(2) If the experiment consists of flipping two coins, then

$$E = \{(H, H), (H, T)\}$$

denotes the event where a head appears on the first coin.

(3) If the experiment consists of tossing a die, then

$$E = \{2, 4, 6\}$$

would be the event that an even number appears on the toss.

For any two events E_1 and E_2 of a sample space S, the *union* of these events $E_1 \cup E_2$ consists of all points, which are either in E_1 or in E_2 or both in E_1 and E_2. Thus, the event

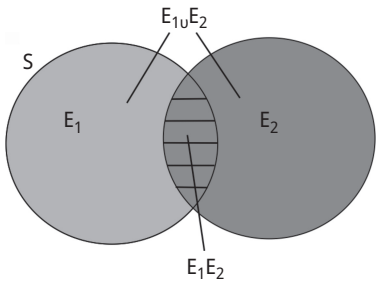


Figure 13.1 Venn diagram showing the sample space S along with the events E_1 and E_2. The union of the two events $E_1 \cup E_2$ is denoted by the grey regions, while the intersection E_1E_2 is depicted in the cross-hatched region where the two events overlap.

$E_1 \cup E_2$ will occur if either E_1 or E_2 occurs. For these same two events E_1 and E_2, we can also define their *intersection* as follows: E_1E_2 consists of all points, which are in both E_1 and E_2, thus requiring that both E_1 and E_2 occur simultaneously. Figure 13.1 gives a simple graphical interpretation of the probability space S along with the events E_1 and E_2 and their union $E_1 \cup E_2$ and intersection E_1E_2.

To again make connection to the sample spaces and event examples given previously, consider the following unions and intersections:

(1) If the experiment consists of flipping a single coin and $E_1 = \{H\}$ and $E_2 = \{T\}$, then

$$E_1 \cup E_2 = \{(H, T)\} = S$$

so that $E_1 \cup E_2$ is the entire sample space S and would occur if the coin flip produces either heads or tails.

(2) If the experiment consists of flipping a single coin and $E_1 = \{H\}$ and $E_2 = \{T\}$ then

$$E_1E_2 = \emptyset$$

where the null event $\emptyset$ cannot occur since the coin cannot be both heads and tails simultaneously. Indeed, if two events are such that $E_1E_2 = \emptyset$, then they are said to be mutually exclusive.

(3) If the experiment consists of tossing a die and if $E_1 = \{1, 3, 5\}$ and $E_2 = \{1, 2, 3\}$, then

$$E_1E_2 = \{1, 3\}$$

so that the intersection of these two events would be a roll of the die of either 1 or 3.

The union and intersection of more than two events can be easily generalized from these ideas. Consider a set of events $E_1, E_2, \cdots, E_N$. Then the union is defined as $\cup_{n=1}^{N} E_n = E_1 \cup E_2 \cup \cdots \cup E_N$ and the intersection is defined as $\prod_{n=1}^{N} E_n = E_1E_2 \cdots E_N$.

Thus, an event which is in any of the E_n belongs to the union, whereas an event needs to be in all E_n in order to be part of the intersection ($n = 1, 2, \cdots, N$). The complement of an event E, denoted E^c, consists of all points in the sample space S not in E. Thus, $E \cup E^c = S$ and $EE^c = \emptyset$.

13.1.2 Probabilities defined on events

With the concept of sample spaces S and events E in hand, the probability of an event $P(E)$ can be calculated. Thus, the following conditions are necessary in assigning a number to $P(E)$:

- $0 \leq P(E) \leq 1$
- $P(S) = 1$
- For any sequence of events $E_1, E_2, \cdots, E_N$ that are mutually exclusive so that $E_i E_j = \emptyset$ when $i \neq j$ then

$$P\left(\cup_{n=1}^{N} E_n\right) = \sum_{n=1}^{N} P(E_n).$$

The notation $P(E)$ is the probability of the event E occurring. The concept of probability is quite intuitive, and it can be quite easily applied to our previous examples.

(1) If the experiment consists of flipping a fair coin, then

$$P(\{H\}) = P(\{T\}) = 1/2$$

where H denotes an outcome of heads and T denotes an outcome of tails. On the other hand, a biased coin that was twice as likely to produce heads over tails would result in

$$P(\{(H\}) = 2/3, \;\; P(\{T\}) = 1/3$$

(2) If the experiment consists of flipping two fair coins, then

$$P(\{(H, H)\} = 1/4$$

where the outcome (H, H) denotes both coins being heads.

(3) If the experiment consists of tossing a fair die, then

$$P(\{1\}) = P(\{2\}) = P(\{3\}) = P(\{4\}) = P(\{5\}) = P(\{6\}) = 1/6$$

and the probability of producing an even number is

$$P(\{2, 4, 6\}) = P(\{2\}) + P(\{4\}) + P(\{6\}) = 1/2.$$

This is a formal definition of the probabilities being functions of events on sample spaces. On the other hand, our intuitive concept of the probability is that if an event is repeated over and over again, then with probability one, the proportion of time an event E occurs is just $P(E)$. This is the frequentist's viewpoint.

A few more facts should be placed here. First, since E and E^c are mutually exclusive and $E \cup E^c = S$, then $P(S) = P(E \cup E^c) = P(E) + P(E^c) = 1$. Additionally, we can compute the formula for $P(E_1 \cup E_2)$, which is the probability that either E_1 or E_2 occurs. From Fig. 13.1 it can be seen that the probability $P(E_1) + P(E_2)$, which represents all points in E_1 and all points in E_2, is given by the grey regions. Notice, in this calculation, that the overlap region is counted twice. Thus, the intersection must be subtracted so that we find

$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 E_2)\,. \tag{13.1.1}$$

If the two events E_1 and E_2 are mutually exclusive, then $E_1 E_2 = \emptyset$ and $P(E_1 \cup E_2) = P(E_1) + P(E_2)$ (note that $P(\emptyset) = 0$).

As an example, consider tossing two fair coins so that each of the points in the sample space $S = \{(H,H),(H,T),(T,H),(T,T)\}$ is equally likely to occur, or said another way, each has probability of $1/4$ to occur. Then let $E_1 = \{(H,H),(H,T)\}$ be the event where the first coin is a head, and let $E_2 = \{(H,H),(T,H)\}$ be the event that the second coin is a head. Then

$$\begin{aligned} P(E_1 \cup E_2) &= P(E_1) + P(E_2) - P(E_1 E_2) \\ &= \frac{1}{2} + \frac{1}{2} - P\{(H,H)\} = \frac{1}{2} + \frac{1}{2} - \frac{1}{4} = \frac{3}{4}. \end{aligned}$$

Of course, this could have also easily been computed directly from the union $P(E_1 \cup E_2) = P(\{(H,H),(H,T),(T,H)\}) = 3/4$.

13.1.3 Conditional probabilities

Conditional probabilities concern, in some sense, the interdependency of events. Specifically, given that the event E_1 has occurred, what is the probability of E_2 occurring? This conditional probability is denoted as $P(E_2|E_1)$. A general formula for the conditional probability can be derived from the following argument: If the event E_1 occurs, then in order for E_2 to occur, it is necessary that the actual occurrence be in the intersection $E_1 E_2$. Thus, in the conditional probability way of thinking, E_1 becomes our new sample space, and the event that $E_1 E_2$ occurs will equal the probability of $E_1 E_2$ relative to the probability of E_1. Thus, we have the conditional probability definition

$$P(E_2|E_1) = \frac{P(E_2 E_1)}{P(E_1)}\,. \tag{13.1.2}$$

The following are examples of some basic conditional probability events.

(1) If the experiment consists of flipping two fair coins, what is the probability that both are heads given that at least one of them is heads? With the sample space $S = \{(H,H),(H,T),(T,H),(T,T)\}$ and each outcome as likely to occur as the other, let E_2 denote the event that both coins are heads, and E_1 denote the event that at least one of them is heads. Then

$$P(E_2|E_1) = \frac{P(E_2E_1)}{P(E_1)} = \frac{P(\{(H,H)\})}{P(\{(H,H),(H,T),(T,H)\})} = \frac{1/4}{3/4} = \frac{1}{3}.$$

(2) Suppose cards numbered one through ten are placed in a hat, mixed and drawn. If we are told that the number on the drawn card is at least five, then what is the probability that it is ten? In this case, E_2 is the probability ($= 1/10$) that the card is a ten, while E_1 is the probability ($= 6/10$) that it is at least a five. Then

$$P(E_2|E_1) = \frac{1/10}{6/10} = \frac{1}{6}\,.$$

13.1.4 Independent events

Independent events will be of significant interest in this book. More specifically, the definition of independent events determines whether or not an event E_1 has any impact, or is correlated, with a second event E_2. By definition, two events are said to be *independent* if

$$P(E_1E_2) = P(E_1)P(E_2)\,. \tag{13.1.3}$$

By (2), this implies that E_1 and E_2 are independent if

$$P(E_2|E_1) = P(E_2)\,. \tag{13.1.4}$$

In other words, the probability of obtaining E_2 does not depend upon whether E_1 occurred. Two events that are not independent are said to be *dependent*.

As an example, consider tossing two fair die where each possible outcome is an equal $1/36$. Then let E_1 denote the event that the sum of the dice is six, E_2 denote the event that the first die equals four and E_3 denote the event that the sum of the dice is seven. Note that

$$\begin{aligned}
P(E_1E_2) &= P(\{(4,2)\}) = 1/36\\
P(E_1)P(E_2) &= 5/36 \times 1/6 = 5/216\\
P(E_3E_2) &= P(\{(4,3)\}) = 1/36\\
P(E_3)P(E_2) &= 1/6 \times 1/6 = 1/36.
\end{aligned}$$

Thus, we can find that E_2 and E_1 cannot be independent, while E_3 and E_2 are, in fact, independent. Why is this true? In the first case where a total of six (E_2) on the die is sought, then the roll of the first die is important as it must be below six in order to satisfy this condition. On the other hand, a total of seven can be accomplished (E_3) regardless of the first die roll.

13.1.5 Bayes's formula

Consider two events E_1 and E_2. Then the event E_1 can be expressed as follows:

$$E_1 = E_1E_2 \cup E_1E_2^c\,. \tag{13.1.5}$$

This is true since in order for a point to be in E_1, it must either be in E_1 and E_2, or it must be in E_1 and not in E_2. Since the intersections E_1E_2 and $E_1E_2^c$ are obviously mutually exclusive, then

$$\begin{aligned} P(E_1) &= P(E_1E_2) + P(E_1E_2^c) \\ &= P(E_1|E_2)P(E_2) + P(E_1|E_2^c)P(E_2^c) \\ &= P(E_1|E_2)P(E_2) + P(E_1|E_2^c)(1 - P(E_2)). \end{aligned} \tag{13.1.6}$$

This states that the probability of the event E_1 is a weighted average of the conditional probability of E_1 given that E_2 has occurred and the conditional probability of E_1 given that E_2 has not occurred, with each conditional probability being given as much weight as the event it is conditioned on has of occurring. To illustrate the application of Bayes's formula, consider the following two examples:

(1) Consider two boxes: the first containing two white and seven black balls, and the second containing five white and six black balls. Now flip a fair coin and draw a ball from the first or second box depending on whether you get heads or tails. What is the conditional probability that the outcome of the toss was heads given that a white ball was selected? For this problem, let W be the event that a white ball was drawn and H be the event that the coin came up heads. The solution to this problem involves the calculation of $P(H|W)$. This can be calculated as follows:

$$\begin{aligned} P(H|W) &= \frac{P(HW)}{P(W)} = \frac{P(W|H)P(H)}{P(W)} \\ &= \frac{P(W|H)P(H)}{P(W|H)P(H) + P(W|H^c)P(H^c)} \\ &= \frac{2/9 \times 1/2}{2/9 \times 1/2 + 5/11 \times 1/2} = \frac{22}{67}. \end{aligned} \tag{13.1.7}$$

(2) A laboratory blood test is 95% effective in determining a certain disease when it is present. However, the test also yields a false positive result for 1% of the healthy persons tested. If 0.5% of the population actually has the disease, what is the probability a person has the disease given that their test result is positive? For this problem, let D be the event that the person tested has the disease, and E the event that their test is positive. The problem involves calculation of $P(D|E)$, which can be obtained by

$$\begin{aligned} P(D|E) &= \frac{P(DE)}{P(E)} \\ &= \frac{P(E|D)P(D)}{P(E|D)P(D) + P(E|D^c)P(D^c)} \\ &= \frac{(0.95)(0.005)}{(0.95)(0.005) + (0.01)(0.995)} \\ &= \frac{95}{294} \approx 0.323 \to 32\%. \end{aligned}$$

In this example, it is clear that if the disease is very rare so that $P(D) \to 0$, it becomes very difficult to actually test for it since even with a 95% accuracy, the chance of the false positives means that the chance of a person actually having the disease is quite small, making the test not so worthwhile. Indeed, if this calculation is redone with the probability that one in a million people of the population have the disease ($P(D) = 10^{-6}$), then $P(D|E) \approx 1 \times 10^{-4} \to 0.01\%$.

Equation (6) can be generalized in the following manner: suppose $E_1, E_2, \cdots, E_N$ are mutually exclusive events such that $\cup_{n=1}^{N} E_n = S$. Thus, exactly one of the events E_n occurs. Further, consider an event H so that

$$P(H) = \sum_{n=1}^{N} P(HE_n) = \sum_{n=1}^{N} P(H|E_n)P(E_n)\,. \tag{13.1.8}$$

Thus, for the given events E_n, one of which must occur, $P(H)$ can be computed by conditioning upon which of the E_n actually occurs. Said another way, $P(H)$ is equal to the weighted average of $P(H|E_n)$, each term being weighted by the probability of the event on which it is conditioned. Finally, using this result, we can compute

$$P(E_n|H) = \frac{P(HE_n)}{P(H)} = \frac{P(H|E_n)P(E_n)}{\sum_{n=1}^{N} P(H|E_n)P(E_n)} \tag{13.1.9}$$

which is known as Bayes's formula.

13.2 **Random Variables and Statistical Concepts**

Ultimately in probability theory, we are interested in the idea of a *random variable*. This is typically the outcome of some experiment that has an undetermined outcome but whose sample space S and potential events E can be characterized. In the examples already presented, the toss of a die and flip of a coin represent two experiments whose outcome is not known until the experiment is performed. A random variable is typically defined as some real-valued function on the sample space. The following are some examples.

(1) Let X denote the random variable, which is defined as the sum of two fair dice. Then the random variable is assigned the values

$$
\begin{aligned}
&P\{X=2\} = P\{(1,1)\} = 1/36\\
&P\{X=3\} = P\{(1,2),(2,1)\} = 2/36\\
&P\{X=4\} = P\{(1,3),(3,1),(2,2,)\} = 3/36\\
&P\{X=5\} = P\{(1,4),(4,1),(2,3),(3,2)\} = 4/36\\
&P\{X=6\} = P\{(1,5),(5,1),(2,4),(4,2),(3,3)\} = 5/36\\
&P\{X=7\} = P\{(1,6),(6,1),(2,5),(5,2),(3,4),(4,3)\} = 6/36\\
&P\{X=8\} = P\{(2,6),(6,2),(3,5),(5,3),(4,4)\} = 5/36\\
&P\{X=9\} = P\{(3,6),(6,3),(4,5),(5,4)\} = 4/36\\
&P\{X=10\} = P\{(4,6),(6,4),(5,5)\} = 3/36\\
&P\{X=11\} = P\{(5,6),(6,5)\} = 2/36\\
&P\{X=12\} = P\{(6,6)\} = 1/36\,.
\end{aligned}
$$

Thus, the random variable X can take on integer values between 2 and 12 with the probability for each assigned above. Since the above events are all possible outcomes that are mutually exclusive, then

$$
P\Big(\cup_{n=2}^{12}\{X=n\}\Big) = \sum_{n=2}^{12} P\{X=n\} = 1.
$$

Figure 13.2 illustrates the probability as a function of X as well as its distribution function.

(2) Suppose a coin is tossed having probability p of coming up heads, until the first head appears. Then define the random variable N, which denotes the number of flips required to generate the first head. Thus,

$$
\begin{aligned}
&P\{N=1\} = P\{H\} = p\\
&P\{N=2\} = P\{(T,H)\} = (1-p)p\\
&P\{N=3\} = P\{(T,T,H)\} = (1-p)^2p\\
&\qquad\vdots\\
&P\{N=n\} = P\{(T,T,\cdots,T,H)\} = (1-p)^{n-1}p
\end{aligned}
$$

and again the random variable takes on discrete values.

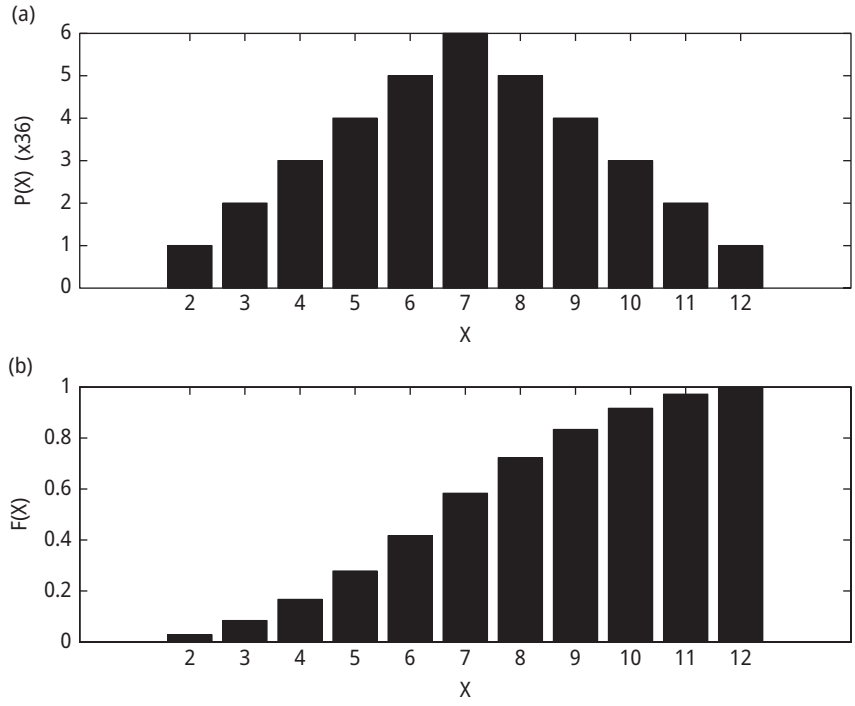


Figure 13.2 (a) Random variable probability assignments $P(X)$ for the sum of two fair dice, and (b) the distribution function $F(X)$ associated with the toss of two fair dice.

In the above examples, the random variables took on a finite, or countable, number of possible values. Such random variables are *discrete* random variables. The cumulative distribution function, or more simply the distribution function $F(\cdot)$ of the random variable X (see Fig. 13.2) is defined for any real number b $(-\infty < b < \infty)$ by

$$F(b) = P\{X \leq b\}\,. \tag{13.2.1}$$

Thus, the $F(b)$ denotes the probability of a random variable X taking on a value, which is less than or equal to b. Some properties of the distribution function are as follows:

- $F(b)$ is a nondecreasing function of b;
- $\lim_{b\to\infty} F(b) = F(\infty) = 1$;
- $\lim_{b\to-\infty} F(b) = F(-\infty) = 0$.

Figure 13.2(b) illustrates these properties quite nicely. Any probability question about X can be answered in terms of the distribution function. The most common and useful example of this is to consider

$$P\{a < X \leq b\} = F(b) - F(a) \tag{13.2.2}$$

for all $a < b$. This result, which gives the probability of $X \in (a, b]$, is fairly intuitive as it involves first computing the probability that $X \leq b\ (F(b))$ and then subtracting from this the probability that $X \leq a\ (F(a))$.

For a discrete random variable, the overall probability function can be defined using the *probability mass function* of X as

$$p(a) = P\{X = a\}\,. \tag{13.2.3}$$

The probability mass function $p(a)$ is positive for at most a countable number of values a so that

$$p(x) = \begin{cases} > 0 & x = x_n\ (n = 1, 2, \cdots, N) \\ = 0 & x \neq x_n. \end{cases} \tag{13.2.4}$$

Further, since X must take on one of the values of x_n, then

$$\sum_{n=1}^{N} p(x_n) = 1\,. \tag{13.2.5}$$

The cumulative distribution is defined as follows:

$$F(a) = \sum_{x_n < a} p(x_n)\,. \tag{13.2.6}$$

In contrast to discrete random variables, *continuous random variables* have an uncountable set of possible values. Many of the definitions and ideas presented thus far for discrete random variables are then easily transferred to the continuous context. Letting X be a continuous random variable, then there exists a nonnegative function $f(x)$ defined for all real $x \in (-\infty, \infty)$ having the property that for any set of real numbers

$$P\{X \in B\} = \int_B f(x)dx\,. \tag{13.2.7}$$

The function $f(x)$ is called the *probability density function* of the random variable X. Thus, the probability that X will be in B is determined by integrating over the set B. Since X must take on some value in $x \in (-\infty, \infty)$, then

$$P\{X \in (-\infty, \infty)\} = \int_{-\infty}^{\infty} f(x)dx = 1\,. \tag{13.2.8}$$

Any probability statement about X can be completely determined in terms of $f(x)$. If we determine the probability that the event $B \in [a, b]$, then

$$P\{a \leq X \leq b\} = \int_a^b f(x)dx\,. \tag{13.2.9}$$

If in this calculation we let $a = b$, then

$$P\{X = a\} = \int_a^a f(x)dx = 0\,. \tag{13.2.10}$$

Thus, the probability that a continuous random variable assumes a *particular* value is zero.

The cumulative distribution function $F(\cdot)$ can also be defined for a continuous random variable. Indeed, it has a simple relationship to the probability density function $f(x)$. Specifically,

$$F(a) = P\{X \in (-\infty, a]\} = \int_{-\infty}^a f(x)dx\,. \tag{13.2.11}$$

Thus, the density is the derivative of the cumulative distribution function. An important interpretation of the density function is obtained by considering the following:

$$P\left\{a - \frac{\epsilon}{2} \le X \le a + \frac{\epsilon}{2}\right\} = \int_{a-\epsilon/2}^{a+\epsilon 2} f(x)dx \approx \epsilon f(a)\,. \tag{13.2.12}$$

The *sifting* property of this integral illustrates that the density function $f(a)$ measures how likely it is that the random variable will be near a.

Having established key properties and definitions concerning discrete and random variables, it is illustrative to consider some of the most common random variables used in practice. Thus, consider first the following discrete random variables:

(1) **Bernoulli random variable** Consider a trial whose outcome can either be termed a success or failure. Let the random variable $X = 1$ if there is a success and $X = 0$ if there is failure. Then the probability mass function of X is given by

$$p(0) = P\{X = 0\} = 1 - p$$
$$p(1) = P\{X = 1\} = p$$

where p ($0 \le p \le 1$) is the probability that the trial is a success.

(2) **Binomial random variable** Suppose n independent trials are performed, each of which results in a success with probability p and failure with probability $1 - p$. Denote X as the number of successes that occur in n such trials. Then X is a binomial random variable with parameters (n, p) and the probability mass function is given by

$$p(j) = \begin{pmatrix} n \\ j \end{pmatrix} p^j (1-p)^{n-j}$$

where $\begin{pmatrix} n \\ j \end{pmatrix} = n!/((n-j)!j!)$ is the number of different groups of j objects that can be chosen from a set of n objects. For instance, if $n = 3$ and $j = 2$, then this binomial

coefficient is 3!/(1!2!) = 3 representing the fact that there are three ways to have two successes in three trials: $(s, s, f), (s, f, s), (f, s, s)$.

(3) **Poisson random variable** A Poisson random variable with parameter λ is defined such that

$$p(n) = P\{X = n\} = e^{-\lambda}\frac{\lambda_n}{n!} \quad n = 0, 1, 2, \cdots$$

for some $\lambda > 0$. The Poisson random variable has an enormous range of applications across the sciences, as it represents, to some extent, the exponential distribution of many systems.

In a similar fashion, a few of the more common continuous random variables can be highlighted here:

(1) **Uniform random variable** A random variable is said to be uniformly distributed over the interval $(0, 1)$ if its probability density function is given by

$$f(x) = \begin{cases} 1, & 0 < x < 1 \\ 0, & \text{otherwise.} \end{cases}$$

More generally, the definition can be extended to any interval (α, β) with the density function being given by

$$f(x) = \begin{cases} \dfrac{1}{\beta - \alpha}, & \alpha < x < \beta \\ 0, & \text{otherwise.} \end{cases}$$

The associated distribution function is then given by

$$F(a) = \begin{cases} 0, & a < \alpha \\ \dfrac{a - \alpha}{\beta - \alpha}, & \alpha < a < \beta \\ 1, & a \geq \beta. \end{cases}$$

(2) **Exponential random variable** Given a positive constant $\lambda > 0$, the density function for this random variable is given by

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & x \leq 0. \end{cases}$$

This is the continuous version of the Poisson distribution considered for the discrete case. Its distribution function is given by

$$F(a) = \int_0^a \lambda e^{-\lambda x} dx = 1 - e^{-\lambda a}, \;\; a > 0\,.$$

(3) **Normal random variable** Perhaps the most important of them all is the normal random variable with parameters μ and σ^2 defined by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}, \quad -\infty < x < \infty.$$

Although we have not defined them yet, the parameters μ and σ refer to the mean and variance of this distribution, respectively. Further, the Gaussian shape produces the ubiquitous bell-shaped curve that is always talked about in the context of large samples, or perhaps grading. Its distribution is given by the error function $erf(a)$, one of the most common functions for look-up tables in mathematics.

13.2.1 Expectation, moments, and variance of random variables

The expected value of a random variable is defined as follows:

$$E[X] = \sum_{x:p(x)>0} xp(x)$$

for a discrete random variable, and

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

for a continuous random variable. Both of these definitions illustrate that the expectation is a weighted average whose weight is the probability that X assumes that value. Typically, the expectation is interpreted as the average value of the random variable. Some examples will serve to illustrate this point:

(1) The expected value $E[X]$ of the roll of a fair die is given by

$$E[X] = 1(1/6) + 2(1/6) + 3(1/6) + 4(1/6) + 5(1/6) + 6(1/6) = 7/2.$$

(2) The expectation $E[X]$ of a Bernoulli variable is

$$E[X] = 0(1-p) + 1(p) = p$$

showing that the expected number of successes in a single trial is just the probability that the trial will be a success.

(3) The expectation $E[X]$ of a binomial or Poisson random variable is

$$E[X] = np \quad \text{binomial}$$
$$E[X] = \lambda \quad \text{Poisson.}$$

(4) The expectation of a continuous uniform random variable is

$$E[X] = \int_{\alpha}^{\beta} \frac{x}{\beta - \alpha} dx = \frac{\alpha + \beta}{2}.$$

(5) The expectation $E[X]$ of an exponential or normal random variable is

$$E[X] = 1/\lambda \quad \text{exponential}$$
$$E[X] = \mu \qquad \text{normal.}$$

The idea of expectation can be generalized beyond the standard expectation $E[X]$. Indeed, we can consider the expectation of any function of the random variable X. Thus, to compute the expectation of some function $g(X)$, the discrete and continuous expectations become:

$$E[g(X)] = \sum_{x:p(x)>0} g(x)p(x) \tag{13.2.13a}$$

$$E[g(X)] = \int_{-\infty}^{\infty} g(x)f(x)dx. \tag{13.2.13b}$$

This also then leads us to the idea of higher moments of a random variable, i.e. $g(X) = X^n$ so that

$$E[X^n] = \sum_{x:p(x)>0} x^n p(x) \tag{13.2.14a}$$

$$E[X^n] = \int_{-\infty}^{\infty} x^n f(x)dx\,. \tag{13.2.14b}$$

The first moment is essentially the mean, the second moment will be related to the variance, and the third and fourth moments measure the *skewness* (asymmetry of the probability distribution) and the *kurtosis* (the flatness or sharpness) of the distribution.

Now that these probability ideas are in place, the quantities of greatest and most practical interest can be considered, namely the ideas of mean and variance (and standard deviation, which is defined as the square root of the variance). The mean has already been defined, while the variance of a random variable is given by

$$Var[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2. \tag{13.2.15}$$

The second form can be easily manipulated from the first. Note that depending upon your scientific community of origin $E[X] = \bar{x} = \langle x \rangle$. It should be noted that the normal distribution has $Var(X) = \sigma^2$, standard deviation σ, and skewness of zero.

13.2.2 Joint probability distribution and covariance

The mathematical framework is now in place to discuss what is perhaps, for this book, the most important use of our probability and statistical thinking, namely the relation between two or more random variables. Ultimately, this assesses whether or not two variables depend upon each other. A joint cumulative probability distribution function of two random variables X and Y is given by

$$F(a,b) = P\{X \le a, Y \le b\}, \;\; -\infty < a,b < \infty\,. \tag{13.2.16}$$

The distribution of X or Y can be obtained by considering

$$F_X(a) = P\{X \le a\} = P\{X \le a, Y \le \infty\} = F(a,\infty) \tag{13.2.17a}$$

$$F_Y(b) = P\{Y \le b\} = P\{Y \le b, X \le \infty\} = F(\infty,b)\,. \tag{13.2.17b}$$

The random variables X and Y are jointly continuous, if there exists a function $f(x,y)$ defined for real x and y so that for any set of real numbers A and B

$$P\{X \in A, Y \in B\} = \int_A \int_B f(x,y)dxdy\,. \tag{13.2.18}$$

The function $f(x,y)$ is called the joint probability density function of X and Y.

The probability density of X or Y can be extracted from $f(x,y)$ as follows:

$$P\{X \in A\} = P\{X \in A, Y \in (-\infty,\infty)\} = \int_{-\infty}^{\infty}\int_A f(x,y)dxdy = \int_A f_X(x)dx \tag{13.2.19}$$

where $f_X(x) = \int_{-\infty}^{\infty} f(x,y)dy$. Similarly, the probability density function of Y is $f_Y(y) = \int_{-\infty}^{\infty} f(x,y)dx$. As previously, to determine a function of the random variables X and Y, then

$$E[g(X,Y)] = \begin{cases} \sum_y \sum_x g(x,y)p(x,y) & \text{discrete} \\ \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)f(x,y)dxdy & \text{continuous.} \end{cases} \tag{13.2.20}$$

Jointly distributed variables lead naturally to a discussion of whether the variables are independent. Two random variables are said to be independent if

$$P\{X \le a, Y \le b\} = P\{X \le a\}P\{Y \le b\}. \tag{13.2.21}$$

In terms of the joint distribution function F of X and Y, independence is established if $F(a,b) = F_X(a)F_Y(b)$ for all a,b. The independence assumption essentially reduces to the following:

$$f(x,y) = f_X(x)f_Y(y)\,. \tag{13.2.22}$$

This further implies

$$E[g(X)h(Y)] = E[g(X)]E[h(Y)]\,. \tag{13.2.23}$$

Mathematically, the case of independence allows for essentially a separation of the random variables X and Y.

What is, to our current purposes, more interesting is the idea of dependent random variables. For such variables, the concept of covariance can be introduced. This is defined as

$$\mathrm{Cov}(X,Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]\,. \tag{13.2.24}$$

If two variables are independent, then $\mathrm{Cov}(X,Y) = 0$. Alternatively, as $\mathrm{Cov}(X,Y)$ approaches unity, it implies that X and Y are essentially directly, or strongly, correlated random variables. This will serve as a basic measure of the statistical dependence of our data sets. It should be noted that although two independent variables implies $\mathrm{Cov}(X,Y) = 0$, the converse is not necessarily true, i.e. $\mathrm{Cov}(X,Y) = 0$ does not imply independence of X and Y.

13.2.3 Python commands

The following are some of the basic commands in Python that we will use for computing the above-mentioned probability and statistical concepts.

(1) **rand(m,n)**: This command generates an $m \times n$ matrix of random numbers from the uniform distribution on the unit interval.

(2) **randn(m,n)**: This command generates an $m \times n$ matrix of normally distributed random numbers with zero mean and unit variance.

(3) **mean(X)**: This command returns the mean or average value of each column of the matrix **X**.

(4) **var(X)**: This computes the variance of each row of the matrix **X**. To produce an unbiased estimator, the variance of the population is normalized by $N - 1$ where N is the sample size.

(5) **std(X)**: This computes the standard deviation of each row of the matrix **X**. To produce an unbiased estimator, the standard deviation of the population is normalized by $N - 1$, where N is the sample size.

(6) **cov(X)**: If **X** is a vector, it returns the variance. For matrices, where each row is an observation, and each column a variable, the command returns a covariance matrix. Along the diagonal of this matrix is the variance of each column.

13.3 Hypothesis Testing and Statistical Significance

With the key concepts of probability theory in hand, statistical testing of data can now be pursued. But before doing so, a few key results concerning a large number of random variables or large number of trials should be explicitly stated.

13.3.1 Limit theorems

Two theorems of probability theory are important to establish before discussing testing of hypothesis via statistics: they are the *strong law of large numbers* and the *central limit theorem*. They are, perhaps, the most well-known and most important practical results in probability theory.

Theorem (Strong law of large numbers): *Let $X_1, X_2, \cdots$ be a sequence of independent random variables having a common distribution, and let $E[X_j] = \mu$. Then, with probability one,*

$$\frac{X_1 + X_2 + \cdots + X_n}{n} \to \mu \text{ as } n \to \infty \tag{13.3.1}$$

As an example of the strong law of large numbers, suppose that a sequence of independent trials is performed. Let E be a fixed event and denote by $P(E)$ the probability that E occurs on any particular trial. Let

$$X_j = \begin{cases} 1, & \text{if } E \text{ occurs on the } j\text{th trial} \\ 0, & \text{if } E \text{ does not occur on the } j\text{th trial.} \end{cases} \tag{13.3.2}$$

The theorem states that with probability one

$$\frac{X_1 + X_2 + \cdots + X_n}{n} \to E[X] = P(E)\,. \tag{13.3.3}$$

Since $X_1 + X_2 + \cdots + X_n$ represents the number of times that the event E occurs in n trials, this result can be interpreted as the fact that, with probability one, the limiting proportion of time that the event E occurs is just $P(E)$.

Theorem (Central Limit Theorem): *Let $X_1, X_2, \cdots$ be a sequence of independent random variables each with mean μ and variance σ^2. Then the distribution of the quantity*

$$\frac{X_1 + X_2 + \cdots + X_n - n\mu}{\sigma\sqrt{n}} \tag{13.3.4}$$

tends to the standard normal distribution as $n \to \infty$ so that

$$P\left\{\frac{X_1 + X_2 + \cdots + X_n - n\mu}{\sigma\sqrt{n}} \le a\right\} \to \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{a} e^{-x^2/2} dx \tag{13.3.5}$$

as $n \to \infty$.

The most important part of this result: it holds for *any* distribution of X_j. Thus, for a large enough sampling of data, i.e. a sequence of independent random variables that goes to infinity, one should observe the ubiquitous bell-shaped probability curve of the normal distribution. This is a powerful result indeed, provided you have a large enough sampling. And

herein lies both its power, and susceptibility to overstating results. The fact is, the normal distribution has some very beautiful mathematical properties, leading us to consider this central limit almost exclusively when testing for statistical significance.

As an example, consider a binomially distributed random variable X with parameters n (n is the number of events) and p (p is the probability of a success of an event). The distribution of

$$\frac{X - E[X]}{\sqrt{Var[X]}} = \frac{X - np}{\sqrt{np(1-p)}} \tag{13.3.6}$$

approaches the standard normal distribution as $n \to \infty$. Empirically, it has been found that the normal approximation will, in general, be quite good for values of n satisfying $np(1-p) \geq 10$.

13.3.2 Statistical decisions

In practice, the power of probability and statistics comes into play in making decisions, or drawing conclusions, based upon a limited sampling of information of an entire *population* (of data). Indeed, it is often impossible or impractical to examine the entire population. Thus, a sample of the population is considered instead. This is called the *sample*. The goal is to infer facts or trends about the entire population with this limited sample size. The process of obtaining samples is called *sampling*, while the process of drawing conclusions about this sample is called *statistical inference*. Using these ideas to make decisions about a population based upon the sample information is termed *statistical decisions*.

In general, these definitions are quite intuitive and also quite well known to us. Statistical inference, for instance, is used extensively in polling data for elections. In such a scenario, county, state, and national election projections are made well before the election based upon sampling a small portion of the population. Statistical decision-making on the other hand is used extensively to determine whether a new medical treatment or drug is, in fact, effective in curing a disease or ailment.

In what follows, the focus will be upon statistical decision-making. In attempting to reach a decision, it is useful to make assumptions or guesses about the population involved. Much like an analysis proof, you can reach a conclusion by assuming something is true and proving it to be so, or assuming the opposite is true and proving that it is false, for instance, by a counter-example. There is no difference with statistical decision-making. In this case, all the statistical tests are hypothesis-driven. Thus, a *statistical hypothesis* is proposed and its validity investigated. As with analysis, some hypotheses are explicitly constructed for the purpose of rejection, or nullifying the hypothesis. For example, to decide whether a coin is fair or loaded, the hypothesis is formulated that the coin is fair with $p = 0.5$ as the probability of heads. To decide if one procedure is better or worse than another, the hypothesis formulated is that there is no difference between the procedures. Such hypotheses are called *null hypotheses* and denoted by H_0. Any hypothesis that differs from a given hypothesis is called an *alternative hypothesis* and is denoted by H_1.

13.3.3 Tests for statistical significance

The ultimate goal in statistical decision-making is to develop tests that are capable of allowing us to accept or reject a hypothesis with a certain, quantifiable confidence level. Procedures that allow us to do this are called *tests of hypothesis, tests of significance*, or *rules of decision*. Overall, they fall within the aegis of the ideas of statistical significance.

In making such statistical decisions, there is some probability that an erroneous conclusion will be reached. Such errors are classified as follows:

- **Type I error**: If a hypothesis is rejected when it should be accepted.
- **Type II error**: If a hypothesis is accepted when it should be rejected.

As an example, suppose a fair coin ($p = 0.5$ for achieving heads or tails) is tossed 100 times and unbelievably, 100 heads in a row results. Any statistical test based upon assuming that the coin is fair is going to be rejected and a Type I error will occur. If, on the other hand, a loaded coin twice as likely to produce a heads than a tails ($p = 2/3$ for achieving a heads and $1 - p = 1/3$ for producing a tails) is tossed 100 times and heads and tails both appear 50 times, then a hypothesis about a biased coin will be rejected and a Type II error will be made.

What is important in making the statistical decision is the *level of significance* of the test. This probability is often denoted by α, and in practice, it is often taken to be 0.05 or 0.01. These correspond to a 5% or 1% level of significance, or alternatively, we are 95% or 99% confident, respectively, that our decision is correct. Basically, in these tests, we will be wrong 5% or 1% of the time.

13.3.4 Hypothesis testing with the normal distribution

For large samples, the central limit theorem asserts that the sample statistics have a normal distribution (or nearly so) with mean μ_s and standard deviation σ_s. Thus, many statistical decision tests are specifically geared to the normal distribution. Consider the normal distribution with mean μ and variance σ^2 such that the density is given by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \quad -\infty < x < \infty \tag{13.3.7}$$

and where the cumulative distribution is given by

$$F(x) = P(X \leq x) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} e^{-(x-\mu)^2/2\sigma^2} dx\,. \tag{13.3.8}$$

With the change of variable

$$Z = \frac{X - \mu}{\sigma} \tag{13.3.9}$$

the *standard normal density function* is given by

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{13.3.10}$$

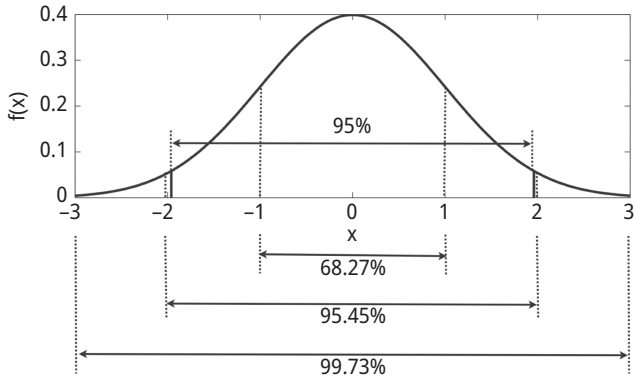


Figure 13.3 Standard normal distribution curve $f(x)$ with the first, second, and third standard deviation markers below the graph along with the corresponding percentage of the population contained within each. Above the graph, markers are placed at $x = -1.96$ and $x = 1.96$ within which 95% of the population is contained. Outside of this range, a statistical test with 95% confidence would result in a conclusion that a given hypothesis is false.

with the corresponding distribution function (related to the error function)

$$F(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-u^2/2} du = \frac{1}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right] . \tag{13.3.11}$$

The standard normal density function has mean zero ($\mu = 0$) and unit variance ($\sigma = 1$). The standard normal curve is shown in Fig. 13.3. In this graph, several key features are demonstrated. On the bottom, the first, second, and third standard deviation markers indicate that 68.72%, 95.45%, and 99.73% of the population are contained within the respective standard deviation markers. Above the graph, a line is drawn at $x = -1.96$ and $x = 1.96$ indicated by the 95% confidence markers. Indeed, the graph indicates two key regions: the *region of acceptance of the hypothesis* for $x \in [-1.96, 1.96]$ and the *region of nonsignificance* for $|x| > 1.96$.

In the above example, interest was given to the extreme values of the statistics, or the corresponding z-value on both sides of the mean. Thus, both tails of the distribution mattered. This is then called a *two-tailed test*. However, some hypothesis tests are only concerned with one side of the mean. Such a test, for example, may be evaluating whether one process is better than another versus whether it is better or worse. This is called a *one-tailed test*. Some examples will serve to illustrate the concepts. It should also be noted that a variety of tests can be formulated for determining confidence intervals, including the Student's t-test (involving measurements on the mean), the chi-square test (involving tests on the variance), and the F-test (involving ratios of variances). These will not be discussed in detail here.

13.3.4.1 Example: A fair coin

Design a decision rule to test the null hypothesis H_0 that a coin is fair if a sample of 64 tosses of the coin is taken, and if the level of significance is 0.05 or 0.01.

The coin toss is an example of a binomial distribution as a success (heads) and failure (tails). Under the null hypothesis that the coin is fair ($p = 0.5$ for achieving either heads or tails), the binomial mean and standard deviation can be easily calculated for n events:

$$\mu = np = 64 \times 0.5 = 32$$
$$\sigma = \sqrt{np(1-p)} = \sqrt{64 \times 0.5 \times 0.5} = 4\,.$$

The statistical decision must now be formulated in terms of the standard normal variable Z where

$$Z = \frac{X - \mu}{\sigma} = \frac{X - 32}{4}\,.$$

For the hypothesis to hold with 95% confidence (or 99% confidence), $z \in [-1.96, 1.96]$. Thus, the following must be satisfied:

$$-1.96 \leq \frac{X - 32}{4} \leq 1.96 \;\rightarrow\; 24.16 \leq X \leq 39.84.$$

Thus, if in a trial of 64 flips, heads appears between 25 and 39 times inclusive, the hypothesis is true with 95% confidence. For 99% confidence, heads must appear between 22 and 42 times inclusive.

13.3.4.2 Example: Class scores

An examination was given to two classes of 40 and 50 students, respectively. In the first class, the mean grade was 74 with a standard deviation of 8, while in the second class the mean was 78 with standard deviation 7. Is there a statistically significant difference between the performance of the two classes at a level of significance of 0.05?

Consider that the two classes come from two populations having respective means μ_1 and μ_2. Then the following null hypothesis and alternative hypothesis can be formulated:

$H_0 : \mu_1 = \mu_2$, and the difference is merely due to chance

$H_1 : \mu_1 \neq \mu_2$, and there is a significant difference between classes.

Under the null hypothesis H_0, both classes come from the same population. The mean and standard deviation of the differences of these is given by

$$\mu_{\bar{X}_1 - \bar{X}_2} = 0$$
$$\sigma_{\bar{X}_1 - \bar{X}_2} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} = \sqrt{\frac{8^2}{40} + \frac{7^2}{50}} = 1.606$$

where the standard deviations have been used as estimates for σ_1 and σ_2. The problem is thus formulated in terms of a new variable representing the difference of the means

$$Z = \frac{\bar{X}_1 - \bar{X}_2}{\sigma_{\bar{X}_1 - \bar{X}_2}} = \frac{74 - 78}{1.606} = -2.49.$$

(a) For a two-tailed test, the results are significant if $|Z| > 1.96$. Thus, it must be concluded that there is a significant difference in performance between the two classes with the second class probably being better.

(b) For the same test with 0.01 significance, the results are significant for $|Z| > 2.58$ so that at a 0.01 significance level, there is no statistical difference between the classes.

13.3.4.3 Example: Rope breaking (Student's t-distribution)

A test of strength of six ropes manufactured by a company showed a breaking strength of 7750 lb and a standard deviation of $145\ lb$, whereas the manufacturer claimed a mean breaking strength of $8000\ lb$. Can the manufacturers claim be supported at a significance level of 0.05 or 0.01?

The decision is between the hypotheses

$$H_0 : \mu = 8000\,\text{lb}, \text{ the manufacturer is justified}$$
$$H_1 : \mu < 8000\,\text{lb}, \text{ the manufacturer's claim is unjustified}$$

where a one-tailed test is to be considered. Under the null hypothesis H_0 and the small sample size, the Student's t-test can be used so that

$$T = \frac{\bar{X} - \mu}{S}\sqrt{n-1} = \frac{7750 - 8000}{145}\sqrt{6-1} = -3.86.$$

As $n \to \infty$, the Student's t-distribution goes to the normal distribution. But in this case of a small sample, the degree of freedom of this distribution is $6 - 1 = 5$, and a 0.05 (0.01) significance is achieved with $T > -2.01$ (or $T > -3.36$). In either case, with $T = -3.86$, it is extremely unlikely that the manufacture's claim is justified.

14 Time–Frequency Analysis: Fourier Transforms and Wavelets

14.1 Basics of Fourier Series and the Fourier Transform 335
- 14.1.1 The Fourier transform 336
- 14.1.2 Derivative relations 337
- 14.1.3 The fast Fourier transform 338
- 14.1.4 Fast Fourier transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT 338
- 14.1.5 FFT versus finite difference differentiation 339
- 14.1.6 Higher-dimensional Fourier transforms 342

14.2 FFT Application: Radar Detection and Filtering 342

14.3 FFT Application: Radar Detection and Averaging 349

14.4 Time–Frequency Analysis: Windowed Fourier Transforms 353
- 14.2.1 Drawbacks of the Gábor (STFT) transform 358
- 14.2.2 Other short-time Fourier transform methods 359

14.5 Time–Frequency Analysis and Wavelets 360
- 14.5.1 Example: The Mexican hat wavelet 364

14.6 Multi-Resolution Analysis and the Wavelet Basis 367
- 14.6.1 Fourier expansions 368
- 14.6.2 Eigenfunction expansions 368
- 14.6.3 Orthonormal wavelets 369
- 14.6.4 Multi-resolution analysis (MRA) 370

14.7 Spectrograms and the Gábor Transform in Python 372
- 14.7.1 JPEG compression with wavelets 378

14.8 Image Processing and Denoising 381
- 14.8.1 Loading images, additive noise, and Python 382

14.9 Diffusion and Image Processing 387
- 14.9.1 The ODE system for 1D diffusion 389
- 14.9.2 2D Python implementation 391
- 14.9.3 Diffusion of an image 391
- 14.9.4 Nonlinear filtering and diffusion 392

14.10 Compressive Sensing and Circumventing Nyquist 393
- 14.10.1 The matrix problem 395
- 14.10.2 Signal reconstruction 396

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0015

Fourier transforms, and more generally, spectral transforms, are one of the most powerful and efficient techniques for solving a wide variety of problems arising in the physical, biological, and engineering sciences. The key idea of the Fourier transform, for instance, is to represent functions and their derivatives as sums of cosines and sines. This operation can be done with the fast Fourier transform (FFT), which is an $O(N \log N)$ operation. Thus, the FFT is faster than most linear solvers of $O(N^2)$. The basic properties and implementation of the FFT will be considered here along with other time–frequency analysis techniques, including wavelets. Such techniques can be broadly applied as will be demonstrated with applications in image processing.

14.1 Basics of Fourier Series and the Fourier Transform

Fourier introduced the concept of representing a given function by a trigonometric series of sines and cosines:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx) \quad x \in (-\pi, \pi]. \tag{14.1.1}$$

At the time, this was a rather startling assertion, especially as the claim held even if the function $f(x)$, for instance, was discontinuous. Indeed, when working with such series solutions, many questions arise concerning the convergence of the series itself. Another, of course, concerns the determination of the expansion coefficients a_n and b_n. Assume for the moment that Eq. (14.1.1) is a uniformly convergent series. Then multiplying by $\cos mx$ gives the following uniformly convergent series:

$$f(x) \cos mx = \frac{a_0}{2} \cos mx + \sum_{n=1}^{\infty} (a_n \cos nx \cos mx + b_n \sin nx \cos mx). \tag{14.1.2}$$

Integrating both sides from $x \in [-\pi, \pi]$ yields the following:

$$\int_{-\pi}^{\pi} f(x) \cos mx dx = \frac{a_0}{2} \int_{-\pi}^{\pi} \cos mx dx + \sum_{n=1}^{\infty} \left(a_n \int_{-\pi}^{\pi} \cos nx \cos mx dx + b_n \int_{-\pi}^{\pi} \sin nx \cos mx dx \right). \tag{14.1.3}$$

But the sine and cosine expressions above are subject to the following *orthogonality* properties:

$$\int_{-\pi}^{\pi} \sin nx \cos mx dx = 0 \quad \forall\ n, m \tag{14.1.4a}$$

$$\int_{-\pi}^{\pi} \cos nx \cos mx dx = \begin{cases} 0 & n \neq m \\ \pi & n = m \end{cases} \tag{14.1.4b}$$

$$\int_{-\pi}^{\pi} \sin nx \sin mx dx = \begin{cases} 0 & n \neq m \\ \pi & n = m. \end{cases} \tag{14.1.4c}$$

Thus, we find the following formulas for the coefficients a_n and b_n:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos nx dx \quad n \geq 0 \tag{14.1.5a}$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin nx dx \quad n > 0 \tag{14.1.5b}$$

which are called the Fourier coefficients.

The expansion (14.1.1) must, by construction, produce 2π-periodic functions since they are constructed from sines and cosines on the interval $x \in (-\pi, \pi]$. In addition to these 2π-periodic solutions, one can consider expanding a function $f(x)$ defined on $x \in (0, \pi]$ as an *even periodic function* on $x \in (-\pi, \pi]$ by employing only the cosine portion of the expansion (14.1.1) or as an *odd periodic function* on $x \in (-\pi, \pi]$ by employing only the sine portion of the expansion (14.1.1).

Using these basic ideas, the complex version of the expansion produces the Fourier series on the domain $x \in [-L, L]$, which is given by

$$f(x) = \sum_{-\infty}^{\infty} c_n e^{in\pi x/L} \quad x \in [-L, L] \tag{14.1.6}$$

with the corresponding Fourier coefficients

$$c_n = \frac{1}{2L} \int_{-L}^{L} f(x) e^{-in\pi x/L} dx. \tag{14.1.7}$$

Although the Fourier series is now complex, the function $f(x)$ is still assumed to be real. Thus, the following observations are of note: (i) c_0 is real and $c_{-n} = c_n*$, (ii) if $f(x)$ is even, all c_n are real, and (iii) if $f(x)$ is odd, $c_0 = 0$, and all c_n are purely imaginary. These results follow from Euler's identity: $\exp(\pm ix) = \cos x \pm i \sin x$. The convergence properties of this representation will not be considered here except that it will be noted that at a discontinuity of $f(x)$, the Fourier series converges to the mean of the left and right values of the discontinuity.

14.1.1 The Fourier transform

The *Fourier transform* is an integral transform defined over the entire line $x \in [-\infty, \infty]$. The Fourier transform and its inverse are defined as

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{14.1.8a}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} F(k) dk\,. \tag{14.1.8b}$$

There are other equivalent definitions. However, this definition will serve to illustrate the power and functionality of the Fourier transform method. We again note that formally, the transform is over the entire real line $x \in [-\infty, \infty]$, whereas our computational domain is only over a finite domain $x \in [-L, L]$. Further, the kernel of the transform, $\exp(\pm ikx)$, describes oscillatory behaviour. Thus, the Fourier transform is essentially an eigenfunction expansion over all continuous wavenumbers k. And once we are on a finite domain $x \in [-L, L]$, the continuous eigenfunction expansion becomes a discrete sum of eigenfunctions and associated wavenumbers (eigenvalues).

14.1.2 Derivative relations

The critical property in the usage of Fourier transforms concerns derivative relations. To see how these properties are generated, we begin by considering the Fourier transform of $f'(x)$. We denote the Fourier transform of $f(x)$ as $\widehat{f(x)}$. Thus, we find

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f'(x) dx = f(x) e^{-ikx}|_{-\infty}^{\infty} + \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx. \tag{14.1.9}$$

Assuming that $f(x) \to 0$ as $x \to \pm\infty$ results in

$$\widehat{f'(x)} = \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx = ik\widehat{f(x)}. \tag{14.1.10}$$

Thus, the basic relation $\widehat{f'} = ik\widehat{f}$ is established. It is easy to generalize this argument to an arbitrary number of derivatives. The final result is the following relation between Fourier transforms of the derivative and the Fourier transform itself:

$$\widehat{f^{(n)}} = (ik)^n \widehat{f}. \tag{14.1.11}$$

This property is what makes Fourier transforms so useful and practical.

As an example of the Fourier transform, consider the following differential equation:

$$y'' - \omega^2 y = -f(x) \quad x \in [-\infty, \infty]. \tag{14.1.12}$$

We can solve this by applying the Fourier transform to both sides. This gives the following reduction:

$$\begin{aligned} \widehat{y''} - \omega^2 \widehat{y} &= -\widehat{f} \\ -k^2 \widehat{y} - \omega^2 \widehat{y} &= -\widehat{f} \\ (k^2 + \omega^2)\widehat{y} &= \widehat{f} \\ \widehat{y} &= \frac{\widehat{f}}{k^2 + \omega^2}. \end{aligned} \tag{14.1.13}$$

To find the solution $y(x)$, we invert the last expression above to yield

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} \frac{\widehat{f}}{k^2 + \omega^2} dk. \tag{14.1.14}$$

This gives the solution in terms of an integral, which can be evaluated analytically or numerically.

14.1.3 The fast Fourier transform

The fast Fourier transform routine was developed specifically to perform the forward and backward Fourier transforms. In the mid-1960s, Cooley and Tukey developed what is now commonly known as the FFT algorithm [40]. Their algorithm was named one of the top ten algorithms of the twentieth century for one reason: the operation count for solving a system dropped to $O(N \log N)$. For N large, this operation count grows almost linearly like N. Thus, it represents a great leap forward from Gaussian elimination and LU decomposition $(O(N^3))$. The key features of the FFT routine are as follows:

(1) It has a low operation count: $O(N \log N)$.

(2) It finds the transform on an interval $x \in [-L, L]$. Since the integration kernel $\exp(ikx)$ is oscillatory, it implies that the solutions on this finite interval have periodic boundary conditions.

(3) The key to lowering the operation count to $O(N \log N)$ is in discretizing the range $x \in [-L, L]$ into 2^n points, i.e. the number of points should be $2, 4, 8, 16, 32, 64, 128, 256, \cdots$.

(4) The FFT has excellent accuracy properties, typically well beyond that of standard discretization schemes.

We will consider the underlying FFT algorithm in detail at a later time. For more information at the present, see [40] for a broader overview.

The practical implementation of the mathematical tools available in Python is crucial. This section will focus on the use of some of the more sophisticated routines in Python, which are cornerstones to scientific computing. Included in this section will be a discussion of the fast Fourier transform routines (*fft, ifft, fftshift, ifftshift, fft2, ifft2*), sparse matrix construction (*spdiag, spy*), and high-end iterative techniques for solving $\mathbf{Ax} = \mathbf{b}$ (*bicgstab, gmres*). These routines should be studied carefully since they are the building blocks of any serious scientific computing code.

14.1.4 Fast Fourier transform: FFT, IFFT, FFTSHIFT, IFFTSHIFT

The fast Fourier transform will be the first subject discussed. Its implementation is straightforward. Given a function that has been discretized with 2^n points and represented by a

vector $\mathbf{x}$, the FFT is found with the command *fft(x)*. Aside from transforming the function, the algorithm associated with the FFT does three major things: it shifts the data so that $x \in [0, L] \to [-L, 0]$ and $x \in [-L, 0] \to [0, L]$, additionally it multiplies every other mode by -1, and it assumes you are working on a 2π-periodic domain. These properties are a consequence of the FFT algorithm discussed in detail at a later time.

To see the practical implications of the FFT, we consider the transform of a Gaussian function. The transform can be calculated analytically so that we have the exact relations:

$$f(x) = \exp(-\alpha x^2) \quad \to \quad \widehat{f}(k) = \frac{1}{\sqrt{2\alpha}} \exp\left(-\frac{k^2}{4\alpha}\right). \tag{14.1.15}$$

A simple Python code to verify this with $\alpha = 1$ is as follows:

```
from numpy.fft import fft, fftshift, ifft

L=20              # define the computational domain [-L/2, L/2]
n=np.power(2,7)   # define the number of Fourier modes 2^n

x = np.linspace(-L/2, L/2, n+1)
x = x[:-1]
u = np.exp(-x*x)

ut = fft(u)                         # FFT the function
utshift = fftshift(ut)              # shift FFT
```

The second figure generated by this script shows how the pulse is shifted. By using the command *fftshift*, we can shift the transformed function back to its mathematically correct positions as shown in the third figure generated. However, before inverting the transformation, it is crucial that the transform is shifted back to the form of the second figure. The command *ifftshift* does this. In general, unless you need to plot the spectrum, it is better not to deal with the *fftshift* and *ifftshift* commands. A graphical representation of the *fft* procedure and its shifting properties is illustrated in Fig. 9.4, where a Gaussian is transformed and shifted by the *fft* routine.

14.1.5 FFT versus finite difference differentiation

Taylor series methods for generating approximations to derivatives of a given function can also be used. In the Taylor series method, the approximations are *local* since they are given by nearest neighbour coupling formulas of finite differences. Using the FFT, the approximation to the derivative is *global* since it uses an expansion basis of cosines and sines, which extend over the entire domain of the given function.

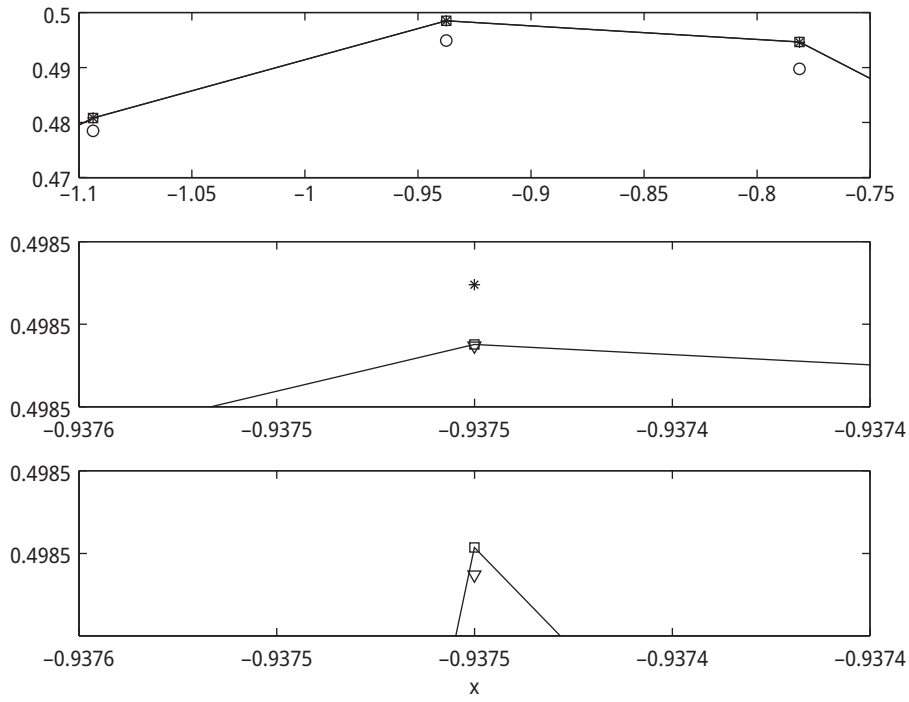


Figure 14.1 Accuracy comparison between second- and fourth-order finite difference methods and the spectral FFT method for calculating the first derivative. Note that by using the *axis* command, the exact solution (line) and its approximations can be magnified near an arbitrary point of interest. Here, the top figure shows that the second-order finite difference (circles) method is within $O(10^{-2})$ of the exact derivative. The fourth-order finite difference (star) is within $O(10^{-5})$ of the exact derivative. Finally, the FFT method is within $O(10^{-6})$ of the exact derivative. This demonstrates the spectral accuracy property of the FFT algorithm.

The calculation of the derivative using the FFT is performed trivially from the derivative relation formula (14.1.11). As an example of how to implement this differentiation formula, consider a specific function:

$$u(x) = \text{sech}(x) \tag{14.1.16}$$

which has the following derivative relations:

$$\frac{du}{dx} = -\text{sech}(x)\tanh(x) \tag{14.1.17a}$$

$$\frac{d^2u}{dx^2} = \text{sech}(x) - 2\,\text{sech}^3(x). \tag{14.1.17b}$$

A comparison can be made between the differentiation accuracy of finite difference formulas of Tables 3.1–3.2 and the spectral method using FFTs (see Fig. 14.1). The following code generates the derivative using the FFT method along with the finite difference $O(\Delta x^2)$ and $O(\Delta x^4)$ approximations considered with finite differences.

```
u = 1/np.cosh(x)      # numpy doesn't have a sech function - use 1/cosh
u1exact = -1 * np.tanh(x) / np.cosh(x)
u2exact = 1/np.cosh(x) - 2/np.power(np.cosh(x), 3)

ut = fft(u)                    # Fourier transform of sech
k = (2*np.pi/L)*np.concatenate((np.arange(0,n/2), np.arange(-n/2,0)))

ut1 = 1j * k * ut                # first derivative in frequency space
ut2 = -k * k * ut                # second derivative in frequency space
u1 = np.real(ifft(ut1))    # inverse transform the first derivative
u2 = np.real(ifft(ut2))    # inverse transform the second derivative

dx = x[1] - x[0]              # dx value needed for finite differences

dbeg = lambda n: -3*u[n] + 4*u[n+1] - u[n+2]  # get appropriate
     diffs at beginning
dend = lambda n: 3*u[n] - 4*u[n-1] + u[n-2]   # get appropriate
     diffs at end

# first derivative finite difference calculation to
     second-order accuracy
ux = np.concatenate(([dbeg(0)], u[2:] - u[:-2], [dend(-1)]))/
    (2*dx)
# ... and to fourth-order accuracy
ux2 = np.concatenate(([dbeg(0), dbeg(1)], -u[4:] + 8*u[3:-1] -
8*u[1:-3] + u[:-4], [dend(-2), dend(-1)]))/(12*dx)
```

Note that the real part is taken after inverse Fourier transforming due to the numerical round-off, which generates a small $O(10^{-15})$ imaginary part.

Of course, this differentiation example works well since the periodic boundary conditions of the FFT are essentially satisfied with the choice of function $f(x) = \mathrm{sech}(x)$. Specifically, for the range of values $x \in [-10, 10]$, the value of $f(\pm 10) = \mathrm{sech}(\pm 10) \approx 9 \times 10^{-5}$. For a function that does not satisfy periodic boundary conditions, the FFT routine leads to significant error. Much of this error arises from the discontinuous jump and the associated

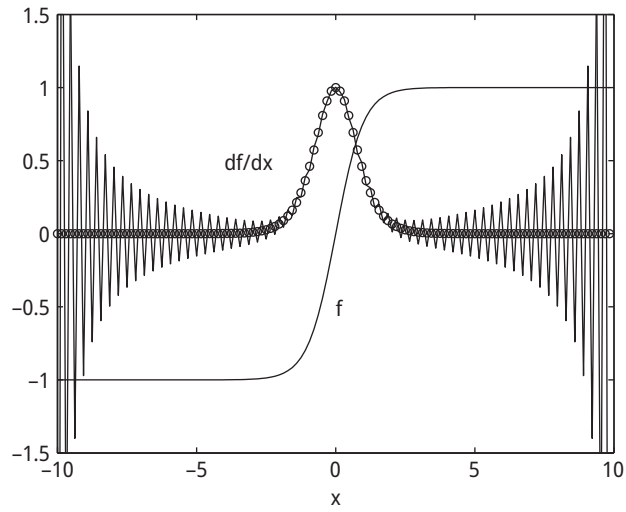


Figure 14.2 Accuracy comparison between the fourth-order finite difference method and the spectral FFT method for calculating the first derivative of $f(x) = \tanh(x)$. The fourth-order finite difference (circle) is within $O(10^{-5})$ of the exact derivative. In contrast, the FFT approximation (line) oscillates strongly and provides a highly inaccurate calculation of the derivative due to the non-periodic boundary conditions of the given function.

Gibb's phenomenon, which results when approximating with a Fourier series. To illustrate this, Fig. 14.2 shows the numerically generated derivative of the function $f(x) = \tanh(x)$, which is $f'(x) = \text{sech}^2(x)$. In this example, it is clear that the FFT method is highly undesirable, whereas the finite difference method performs as well as before.

14.1.6 Higher-dimensional Fourier transforms

For transforming in higher dimensions, a couple of choices in Python are possible. For 2D transformations, it is recommended to use the commands **fft2** and **ifft2**. These will transform a matrix $\mathbf{A}$, which represents data in the x- and y-direction, along the rows and then columns, respectively. For higher dimensions, the **fft** command can be modified to **fft(x,[],N)**, where N is the number of dimensions. Alternatively, use **fftn(x)** for multi-dimensional FFTs.

14.2 FFT Application: Radar Detection and Filtering

FFTs and other related frequency transforms have revolutionized the field of digital signal processing and imaging. The key concept in any of these applications is to use the FFT to analyse and manipulate data in the frequency domain. There are other methods of treating

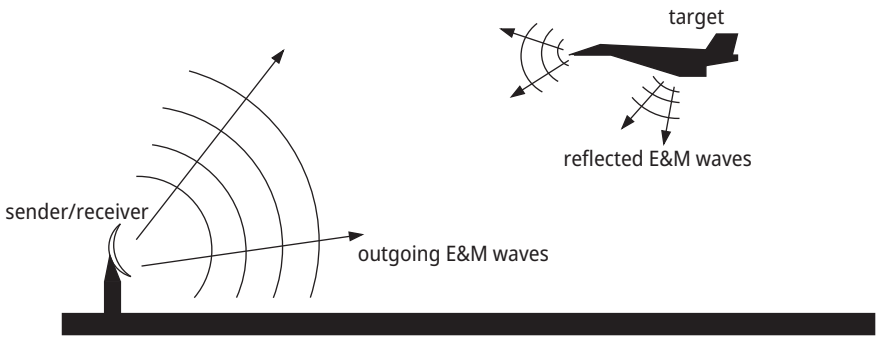


Figure 14.3 Schematic of the operation of a radar detection system. The radar emits an electromagnetic (E&M) field at a specific frequency. The radar then receives the reflection of this field from objects in the environment and attempts to determine what the objects are.

the signal in the time and frequency domain, but the simplest to begin with is the Fourier transform.

The Fourier transform is also used in quantum mechanics to represent a quantum wavepacket in either the spatial domain or the momentum (spectral) domain. Quantum mechanics is specifically mentioned due to the well-known *Heisenberg uncertainty principle*. Simply stated, you cannot know the exact position and momentum of a quantum particle simultaneously. This is simply a property of the Fourier transform itself. Essentially, a narrow signal in the time domain corresponds to a broadband source, whereas a highly confined spectral signature corresponds to a broad time-domain source. This has significant implication for many techniques of signal analysis. In particular, an excellent decomposition of a given signal into its frequency components can render no information about *when* in time the different portions of signal actually occurred. Thus, there is often a competition between trying to localize both in time and frequency a signal or stream of data. Time–frequency analysis is ultimately all about trying to resolve the time and frequency domain in an efficient and tractable manner. Various aspects of this time–frequency processing will be considered in later sections, including wavelet-based methods of time–frequency analysis.

At this point, we discuss a very basic concept and manipulation procedure to be performed in the frequency domain: noise attenuation via frequency (band-pass) filtering. This filtering process is fairly common in electronics and signal detection. As a specific application or motivation for spectral analysis, consider the process of radar detection depicted in Fig. 14.3. An outgoing electromagnetic field is emitted from a source, which then attempts to detect the reflections of the emitted signal. In addition to reflections coming from the desired target, reflections can also come from geographical objects (mountains, trees, etc.) and atmospheric phenomena (clouds, precipitation, etc.). Further, there may be other sources of the electromagnetic energy at the desired frequency. Thus, the radar detection process can be greatly complicated by a wide host of environmental phenomena. Ultimately, these effects combine to give at the detector a noisy signal, which must be processed and filtered before a detection diagnosis can be made.

It should be noted that the radar detection problem discussed in what follows is highly simplified. Indeed, modern day radar detection systems use much more sophisticated time–frequency methods for extracting both the position and spectral signature of target objects. Regardless, this analysis gives a high-level view of the basic and intuitive concepts associated with signal detection.

To begin, we consider an ideal signal, i.e. an electromagnetic pulse, generated in the time domain as shown in the last code block. This will generate an ideal hyperbolic secant shape in the time domain. It is this time-domain signal that we will attempt to reconstruct via denoising and filtering. Figure 14.4(a) illustrates the ideal signal.

In most applications, the signals as generated above are not ideal. Rather, they have a large amount of noise integrated within them. Usually, this noise is what is called *white noise*, i.e. a noise that affects all frequencies the same. We can add white noise to this signal by considering the pulse in the frequency domain.

```
noise = 1
ut = fft(u)
utn = ut + noise*(randn(n) + (1j)*randn(n))
un = ifft(utn)
```

These lines of code generate the Fourier transform of the function along with the vector **utn**, which is the spectrum of the given signal with a complex and Gaussian distributed (mean zero, unit variance) noise source term added in. Figure 14.4 shows the difference between the ideal time-domain signal pulse and the more physically realistic pulse for which white noise has been added. In these figures, a clear *signal*, i.e. the original time-domain pulse, is still detected even in the presence of noise.

The fundamental question in signal detection now arises: is the time-domain structure received in a detector simply a result of noise, or is there an underlying signal being transmitted? For small amounts of noise, this is clearly not a problem. However, for large amounts of noise or low levels of signal, the detection becomes a nontrivial matter.

In the context of radar detection, two competing objectives are at play: for the radar detection system, an accurate diagnosis of the data is critical in determining the presence of an aircraft. Competing in this regard is, perhaps, the aircraft's objective of remaining invisible, or undetected, from the radar. Thus, an aircraft attempting to remain invisible will attempt to reflect as little electromagnetic signal as possible. This can be done by, for instance, covering the aircraft with materials that absorb the radar frequencies used for detection, i.e. stealth technology. An alternative method is to fly another aircraft through the region, which bombards the radar detection system with electromagnetic energy at the detection frequencies. Thus, the aircraft, attempting to remain undetected, may be successful since the radar system may not be able to distinguish between the sources of the electromagnetic fields. This second option is, in some sense, like a radar jamming system. In the first case, the radar detection system will have a small signal-to-noise ratio,

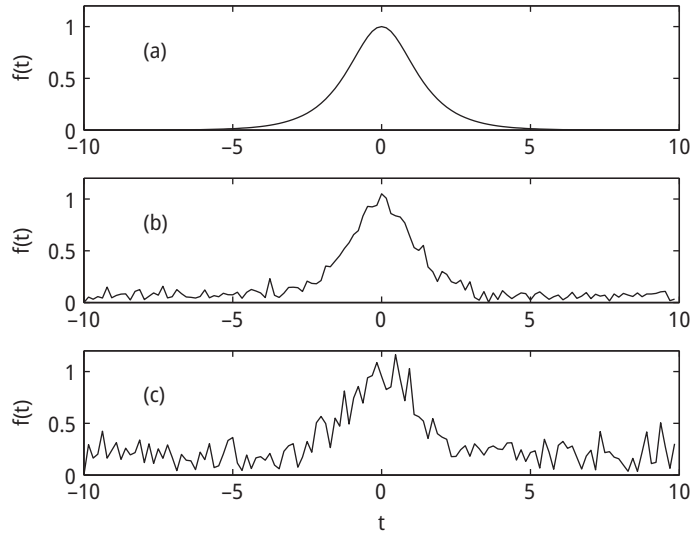


Figure 14.4 Ideal time-domain pulse (a) along with two realizations of the pulse with increasing noise strength (b) and (c). The noisy signals are what are typically expected in applications.

and denoising will be critical to accurate detection. In the second case, an accurate time–frequency analysis is critical for determining the time localization and position of the competing signals.

The focus of what follows is to apply the ideas of spectral filtering to attempt to improve the signal detection by denoising. Spectral filtering is a method, which allows us to extract information at specific frequencies. For instance, in the radar detection problem, it is understood that only a particular frequency (the emitted signal frequency) is of interest at the detector. Thus, it would seem reasonable to filter out, in some appropriate manner, the remaining frequency components of the electromagnetic field received at the detector.

Consider a very noisy field created around the hyperbolic secant of the previous example (see Fig. 14.5). Specifically, we take the noise=10 in the last example code block. Figure 14.5 demonstrates the impact of a large noise applied to the underlying signal. In particular, the signal is completely buried within the white noise fluctuations, making the detection of a signal difficult, if not impossible, with the unfiltered noisy signal field.

Filtering can help significantly improve the ability to detect the signal buried in the noisy field of Fig. 14.5. For the radar application, the frequency (wavenumber) of the emitted and reflected field is known, thus spectral filtering around this frequency can remove undesired frequencies and much of the white noise picked up in the detection process. There are a wide range of filters that can be applied to such a problem. One of the simplest filters to consider here is the Gaussian filter (see Fig. 14.6):

$$\mathcal{F}(k) = \exp\left(-\tau(k-k_0)^2\right) \tag{14.2.1}$$

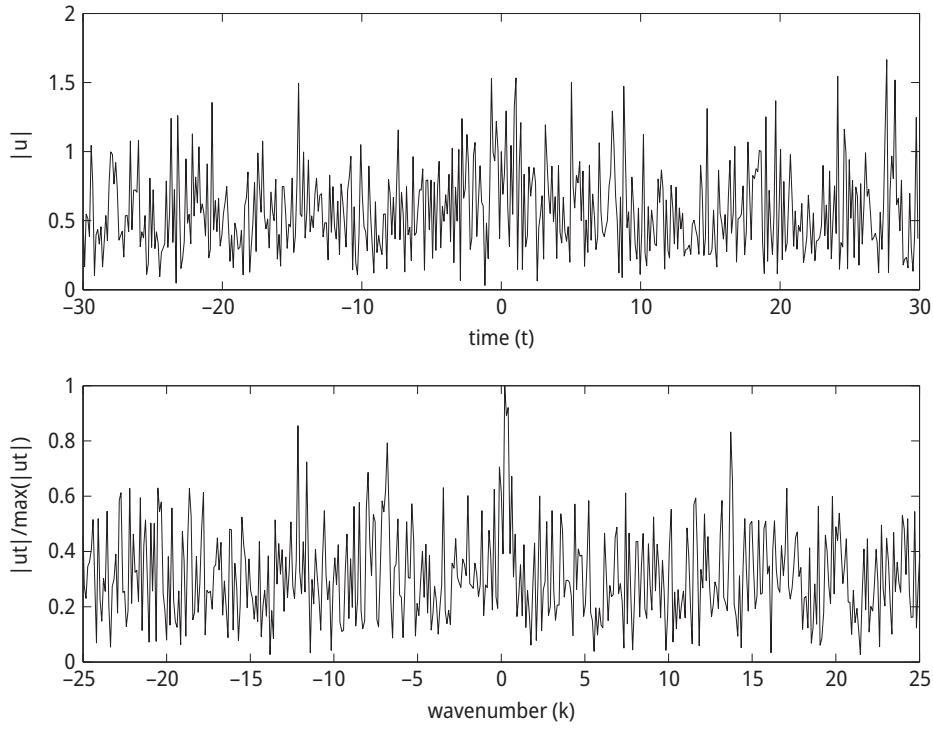


Figure 14.5 Time-domain (top) and frequency-domain (bottom) plots for a single realization of white noise. In this case, the noise strength has been increased to ten, thus burying the desired signal field in both time and frequency.

where τ measures the bandwidth of the filter, and k is the wavenumber. The generic filter function $\mathcal{F}(k)$ in this case acts as a low-pass filter since it eliminates high-frequency components in the system of interest. Note that these are high frequencies in relation to the centre frequency $(k = k_0)$ of the desired signal field. In the example considered here $k_0 = 0$.

Application of the filter strongly attenuates those frequencies away from the centre frequency k_0. Thus, if there is a signal near k_0, the filter isolates the signal input around this frequency or wavenumber. Application of the filtering results in the following spectral processing in Python:

```
bw = 0.2
filter = np.exp(-bw*np.power(k,2))
unft = filter*unt    # multiply the filter by the signal in
   the frequency domain
unf = ifft(unft)      # invert the transform
```

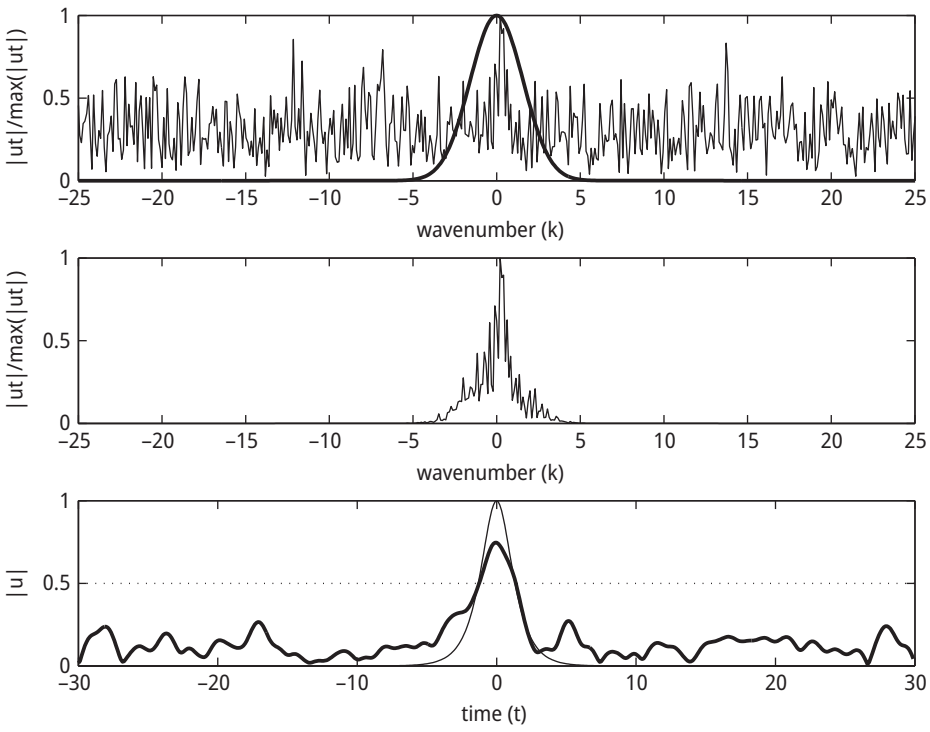


Figure 14.6 (top) White-noise inundated signal field in the frequency domain along with a Gaussian filter with bandwidth parameter $\tau = 0.2$ centred on the signal centre frequency. (middle) The post-filtered signal field in the frequency domain. (bottom) The time-domain reconstruction of the signal field (bolded line) along with the ideal signal field (light line) and the detection threshold of the radar (dotted line).

The results of this code are illustrate in Fig. 14.6, where the white noise inundated signal field in the spectral domain is filtered with a Gaussian filter centred around the zero wavenumber $k_0 = 0$ with a bandwidth parameter $\tau = 0.2$. The filtering extracts nicely the signal field despite the strength of the applied white noise. Indeed, Fig. 14.6 (bottom) illustrates the effect of the filtering process and its ability to reproduce an approximation to the signal field. In addition to the extracted signal field, a detection threshold is placed (dotted line) on the graph in order to illustrate that the detector would read the extracted signal as a target.

As a matter of completeness, the extraction of the electromagnetic field is also illustrated when not centred on the centre frequency. Figure 14.7 shows the field that is extracted for a filter centred around $k_0 = 15$. This is done easily with the Python commands

```
k0 = 15; bw=0.2
unft = np.exp(-bw*np.power(k-k0,2))*unt
unf = ifft(unft)
```

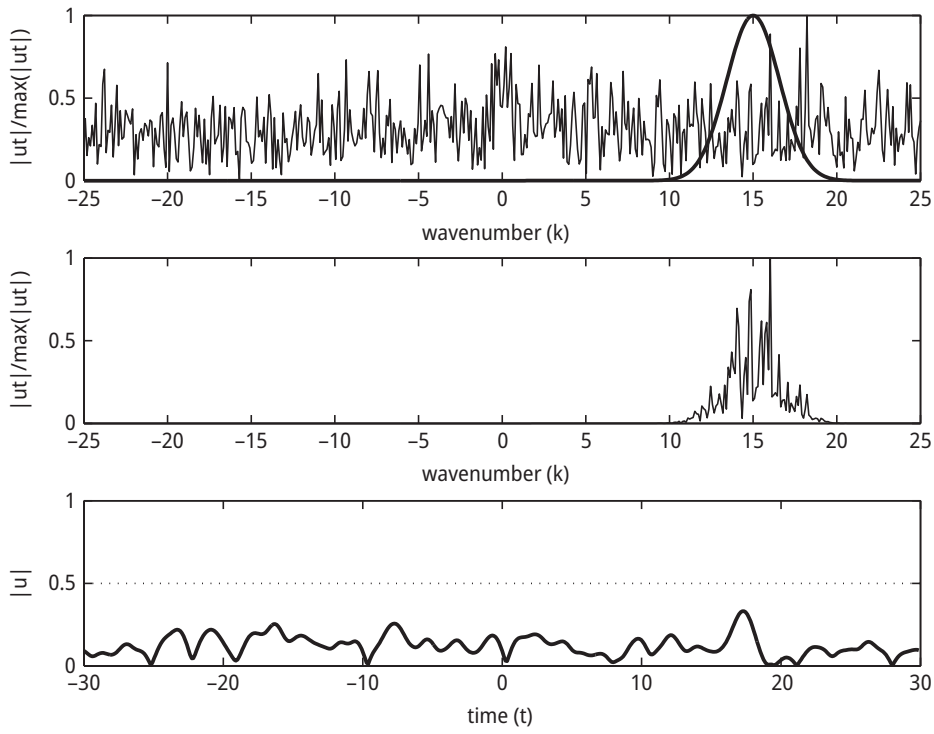


Figure 14.7 (top) White-noise inundated signal field in the frequency domain along with a Gaussian filter with bandwidth parameter $\tau = 0.2$ centred on the signal frequency $k = 15$. (middle) The post-filtered signal field in the frequency domain. (bottom) The time-domain reconstruction of the signal field (bolded line) along the detection threshold of the radar (dotted line).

In this case, there is no signal around the wavenumber $k_0 = 15$. However, there is a great deal of white noise electromagnetic energy around this frequency. Upon filtering and inverting the Fourier transform, it is clear from Figure 14.7 (bottom) that there is no discernible target present. As before, a decision threshold is set at $|u| = 0.5$, and the white noise fluctuations clearly produce a field well below the threshold line.

In all of the analysis above, filtering is done on a given set of signal data. However, the signal is evolving in time, i.e. the position of the aircraft is probably moving. Moreover, one can imagine that there is some statistical chance that a type I or type II error can be made in the detection. Thus, a target may be identified when there is no target, or a target has been missed completely. Ultimately, the goal is to use repeated measurements from the radar of the airspace in order to try to avoid a detection error or failure. Given that you may launch a missile, the stakes are high and a high level of confidence is needed in the detection process. It should also be noted that filter design, shape, and parametrization play significant roles in designing optimal filters. Thus, filter design is an object of strong interest in signal processing applications.

14.3 FFT Application: Radar Detection and Averaging

The previous section clearly shows the profound and effective use of filtering in cleaning up a noisy signal. This is one of the most basic illustrations of the power of basic signal processing. Other methods also exist to help reduce noise and generate better time–frequency resolution. In particular, the filtering example given fails to use two key facts: the noise is white, and radar will continue to produce more signal data. These two facts can be used to produce useful information about the incoming signal. In particular, it is known, and demonstrated in the previous section, that white noise can be modelled adding a normally distributed random variable with zero mean and unit variance to each Fourier component of the spectrum. The key here: *with zero mean*. Thus, over many signals, the white noise should, on average, add up to zero; a very simple concept, yet tremendously powerful in practice for signal processing systems where there is continuous detection of a signal.

To illustrate the power of denoising the system by averaging, the simple analysis of the previous section will be considered for a time pulse. In beginning this process, a time window must be selected and its Fourier resolution decided. In order to be more effective in plotting our spectral results, the Fourier components in the standard shifted frame ($\mathbf{k}$) and its unshifted counterpart ($\mathbf{ks}$) are both produced. Further, the noise strength of the system is selected.

In the next portion of the code, the total number of data frames, or realizations of the incoming signal stream, is considered. The word realizations is used here to help underscore the fact that for each data frame, a new realization of the white noise is produced. In the following code, 1, 2, 5, and 100 realizations are considered.

```
realize = [1, 2, 5, 100]

for r in np.arange(0,len(realize)):
    ii = realize[r]
    utn = np.zeros((ii,n), dtype=complex) # store the transform
    un = np.zeros((ii,n), dtype=complex)  # store the inverse transform
    dat = np.zeros((ii,n), dtype=complex) # initialize the data
    for i in np.arange(0,ii):
        utn[i,:] = ut + noise*(randn(1,n) + 1j*randn(1,n)) # add noise
        dat[i,:] = np.abs(fftshift(utn[i,:]))/np.max(np. abs(utn[i,:]))
        un[i,:] = ifft(utn[i,:])
    ave = np.abs(fftshift(utn.sum(0)))/ii # average over realizations
```

Figure 14.8 demonstrates the averaging process in the spectral domain as a function of the number of realizations. With one realization, it is difficult to determine if there is a true signal, or if the entire spectrum is noise. Already after two realizations, the centre frequency

structure starts to appear. Five realizations has a high degree of discrimination between the noise and signal and 100 realizations is almost exactly converged to the ideal, noise-free signal. As expected, the noise in the averaging process is eliminated since its mean is zero. To view a few of the data realizations used in computing the averaging in Fig. 14.8, the first eight signals used to compute the average for 100 realizations are shown in Fig. 14.9.

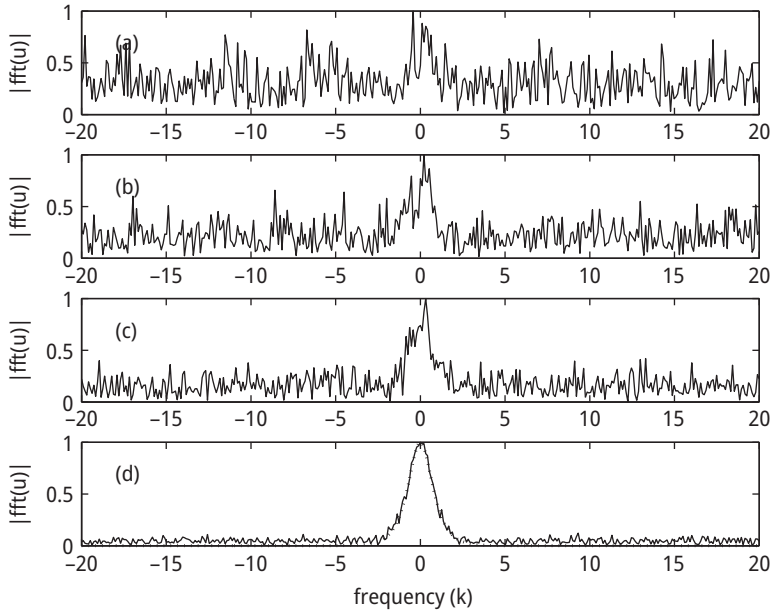


Figure 14.8 Average spectral content for (a) 1, (b) 2, (c) 5, and (d) 100 realizations of the data. The dotted line in (d) represents the ideal, noise-free spectral signature.

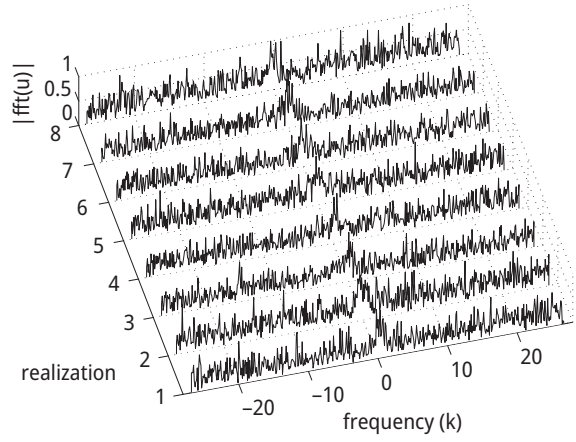


Figure 14.9 Typical time-domain signals that would be produced in a signal detection system. At different times of measurement, the added white-noise would alter the signal stream significantly.

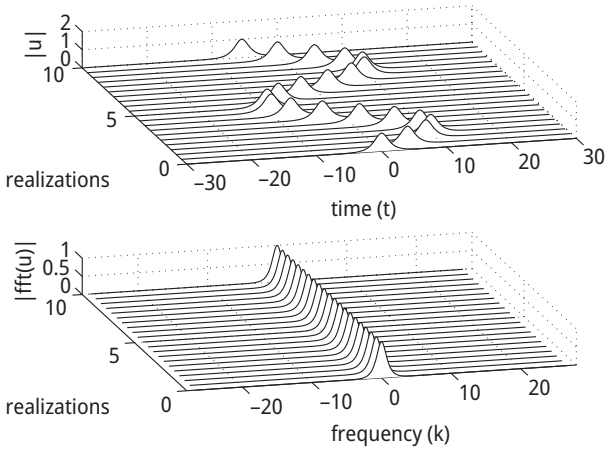


Figure 14.10 Ideal time–frequency behaviour for a time-domain pulse that evolves dynamically in time. The spectral signature remains unchanged as the pulse moves over the time domain.

Figures 14.8 and 14.9 illustrate how the averaging process can ultimately extract a clean spectral signature. As a consequence, however, you completely loose the time-domain dynamics. In other words, one could take the cleaned up spectrum of Fig. 14.8 and invert the Fourier transform, but this would only give the averaged time-domain signal but not how it actually evolved over time. To consider this problem more concretely, consider the following bit of code that generates a time-varying signal shown in Fig. 14.10. There are a total of 21 realizations of data in this example.

```
slice = np.arange(0,10.1,0.5)     # time slices
[T, S] = np.meshgrid(t, slice)
[K, S] = np.meshgrid(k, slice)
freq = 0          # center frequency (k) of signal
U = 1/np.cosh(T-10*np.sin(S))*np.exp(1j*freq*T)

Kp = fftshift(K,1)      # shift within realization (on the 1 axis)
Ut = fft(U)          # clean signal in frequency domain

Utn = Ut + noise*(randn(*Ut.shape) + 1j*randn(*Ut.shape))    # noise
Un = ifft(Utn)        # noisy signal in time domain
Utn_rnorm = np.abs(Utn).max(1).reshape(len(slice),1)
Utnp = fftshift(Utn,1)/Utn_rnorm  # noisy, shifted, normalized freq
    domain
```

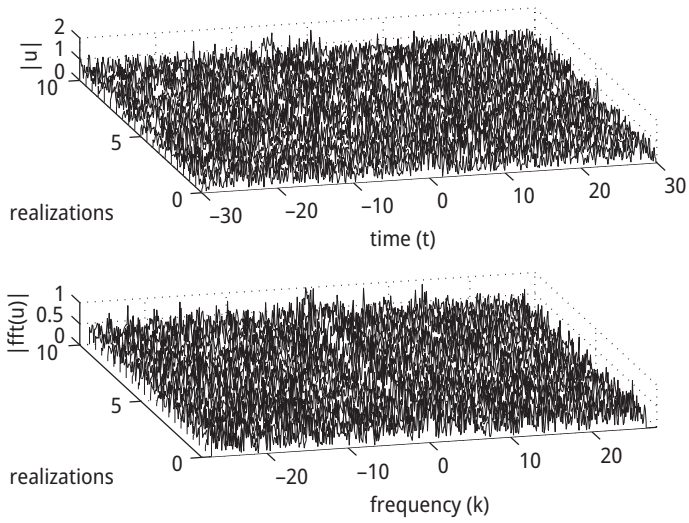


Figure 14.11 A more physically realistic depiction of the time domain and frequency spectrum as a function of the number of realizations. The goal is to extract the meaningful data buried within the noise.

If this were a radar problem, the movement of the signal would represent an aircraft moving in time. The spectrum, however, remains fixed at the transmitted signal frequency. Thus, it is clear even at this point that averaging over the frequency realizations produces a clean, localized signature in the frequency domain, whereas averaging over the time domain only smears out the evolving time-domain pulse.

The ideal pulse evolution in Fig. 14.10 is now inundated with white noise as shown in Fig. 14.11. The noise is added to each realization, or data measurement, with the same strength as shown in Figs. 14.8 and 14.9. The obvious question arises about how to clean up the signal and whether something meaningful can be extracted from the data or not. After all, there are only 21 data slices to work with. The following code averages over the 21 data realizations in both the time and frequency domains.

```
Uavg = Un.sum(0)/len(slice)
Utavg = fftshift(Utn.sum(0))/len(slice)
```

Figure 14.12 shows the results from the averaging process for a nonstationary signal. The top graph shows that averaging over the time domain for a moving signal produces no discernible signal. However, averaging over the frequency domain produces a clear signature at the centre frequency of interest. Ideally, if more data is collected, a better average signal is produced. However, in many applications, the acquisition of data is limited and decisions must be made upon what are essentially small sample sizes.

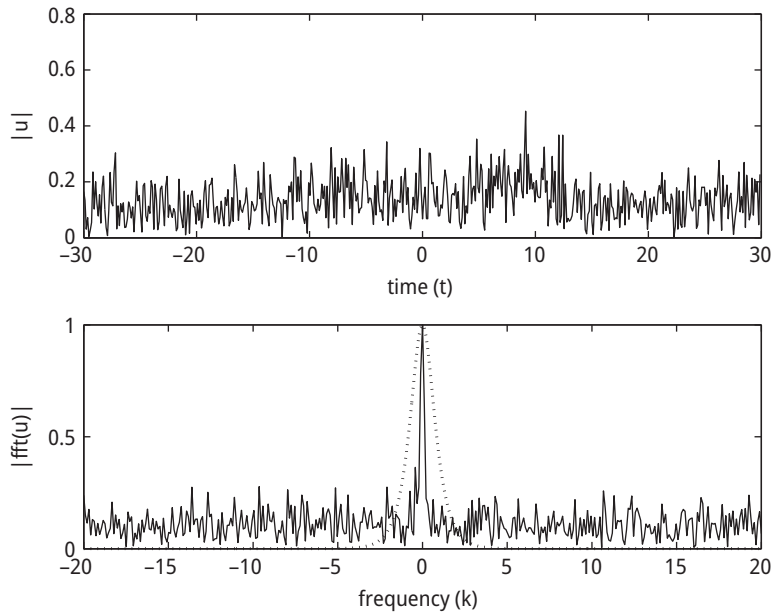


Figure 14.12 Averaged time-domain and spectral profiles for the 21 realizations of data shown in Fig. 14.11. Even with a limited sampling, the spectral signature at the centre frequency is extracted.

14.4 Time–Frequency Analysis: Windowed Fourier Transforms

The Fourier transform is one of the most important and foundational methods for the analysis of signals. However, it was realized very early on that Fourier transform-based methods had severe limitations. Specifically, when transforming a given time signal, it is clear that all the frequency content of the signal can be captured with the transform, but the transform fails to capture the *moment in time* when various frequencies were actually exhibited. Figure 14.13 shows a prototypical signal that may be of interest to study. The signal $S(t)$ clearly comprises various frequency components that are exhibited at different times. For instance, at the beginning of the signal, there are high-frequency components. In contrast, the middle of the signal has relatively low-frequency oscillations. If the signal represented music, then the beginning of the signal would produce high notes, while the middle would produce low notes. The Fourier transform of the signal contains all this information, but there is no indication of *when* the high or low notes actually occur in time. Indeed, by definition the Fourier transform eliminates all time-domain information since you actually integrated out all time in (14.1.8b).

The obvious question to arise is this: What is the Fourier transform good for in the context of signal processing? In the previous sections where the Fourier transform was applied, the signal being investigated was fixed in frequency, i.e. a sonar or radar detector with a

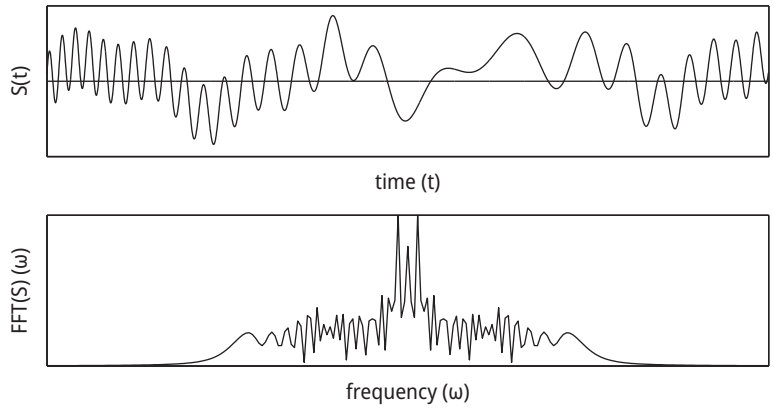


Figure 14.13 Signal $S(t)$ and its normalized Fourier transform $\hat{S}(\omega)$. Note the large number of frequency components that make up the signal.

fixed frequency ω_0. Thus for a given signal, the frequency of interest did not shift in time. By using different measurements in time, a signal-tracking algorithm could be constructed. Thus, an implicit assumption was made about the invariance of the signal frequency. Ultimately, the Fourier transform is superb for one thing: characterizing *stationary* or periodic signals. Informally, a stationary signal is such that repeated measurements of the signal in time yield an average value that does not change in time. Most signals, however, do not satisfy this criterion. A song, for instance, changes its average Fourier components in time as the song progresses in time. Thus, the generic signal $S(t)$ that should be considered, and that is plotted as an example in Fig. 14.13, is a *nonstationary signal* whose average signal value does change in time. It should be noted that in our application of radar detection of a moving target, use was made of the stationary nature of the spectral content. This allowed for a clear idea of where to filter the signal $\hat{S}(\omega)$ in order to reconstruct the signal $S(t)$.

Having established the fact that the direct application of the Fourier transform provides an untenable method for extracting signal information, it is natural to pursue modifications of the method in order to extract time and frequency information. The most simple-minded approach is to consider Fig. 14.13 and to decompose the signal over the time domain into separate time frames. Figure 14.14 shows the original signal $S(t)$ considered but now decomposed into four smaller time windows. In this decomposition, for instance, the first time frame is considered with the remaining three time frames zeroed out. For each time window, the Fourier transform is applied in order to characterize the frequencies present during that time frame. The highest frequency components are captured in Fig. 14.14(a), which is clearly seen in its Fourier transform. In contrast, the slow modulation observed in the third time frame (c) is devoid of high-frequency components as observed in Fig. 14.14(c). This method thus exhibits the ability of the Fourier transform, appropriately modified, to extract out both time and frequency information from the signal.

The limitations of the direct application of the Fourier transform, and its inability to localize a signal in both the time and frequency domains, were realized very

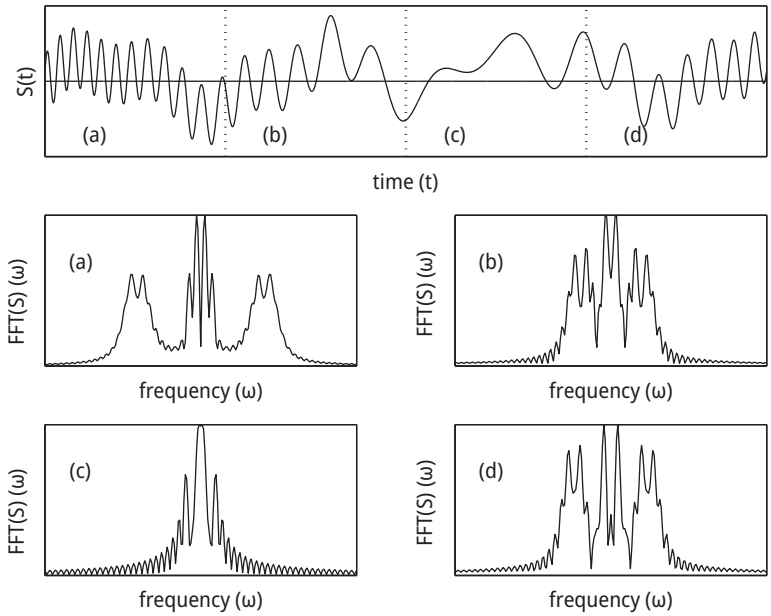


Figure 14.14 Signal $S(t)$ decomposed into four equal and separate time frames (a), (b), (c), and (d). The corresponding normalized Fourier transform of each time frame $\hat{S}(\omega)$ is illustrated below the signal. Note that this decomposition gives information about the frequencies present in each smaller time frame.

early on in the development of radar and sonar detection. The Hungarian physicist/mathematician/electrical engineer Gábor Dénes (Nobel Prize for Physics in 1971 for the discovery of holography in 1947) was the first to propose a formal method for localizing both time and frequency. His method involved a simple modification of the Fourier transform kernel. Thus, Gábor introduced the kernel

$$g_{t,\omega}(\tau) = e^{i\omega\tau} g(\tau - t) \tag{14.4.1}$$

where the new term to the Fourier kernel $g(\tau - t)$ was introduced with the aim of localizing both time and frequency. The *Gábor transform*, also known as the *short-time Fourier transform (STFT)*, is then defined as the following:

$$\mathcal{G}[\,f](t,\omega) = \tilde{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau)\bar{g}(\tau - t)e^{-i\omega\tau} d\tau = (\,f, \bar{g}_{t,\omega}) \tag{14.4.2}$$

where the bar denotes the complex conjugate of the function. Thus, the function $g(\tau - t)$ acts as a time filter for localizing the signal over a specific window of time. The integration over the parameter τ slides the time-filtering window down the entire signal in order to pick out the frequency information at each instant of time. Figure 14.15 gives a nice illustration of how the time-filtering scheme of Gábor works. In this figure, the time-filtering window is centred at τ with a width a. Thus, the frequency content of a window of time is extracted and

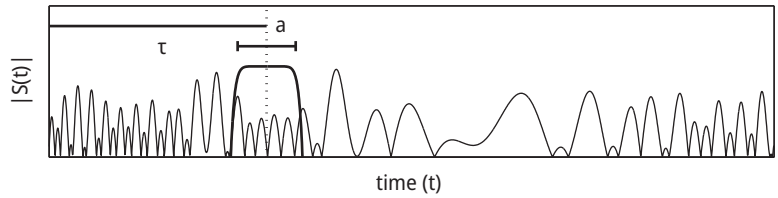


Figure 14.15 Graphical depiction of the Gábor transform for extracting the time–frequency content of a signal $S(t)$. The time-filtering window $g(\tau - t)$ is centred at τ with width a.

τ is modified to extract the frequencies of another window. The definition of the Gábor transform captures the entire time–frequency content of the signal. Indeed, the Gábor transform is a function of the two variables t and ω.

A few of the key mathematical properties of the Gábor transform are highlighted here. To be more precise about these mathematical features, some assumptions about commonly used $g_{t,\omega}$ are considered. Specifically, for convenience, we will consider g to be real and symmetric with $\|g(t)\| = 1$ and $\|g(\tau - t)\| = 1$, where $\|\cdot\|$ denotes the L_2 norm. Thus, the definition of the Gábor transform, or STFT, is modified to

$$\mathcal{G}[f](t,\omega) = \tilde{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau)g(\tau - t)e^{-i\omega\tau}d\tau \tag{14.4.3}$$

with $g(\tau - t)$ inducing localization of the Fourier integral around $t = \tau$. With this definition, the following properties hold:

(1) The energy is bounded by the Schwarz inequality so that

$$|\tilde{f}_g(t,\omega)| \leq \|f\|\|g\|. \tag{14.4.4}$$

(2) The energy in the signal plane around the neighbourhood of (t,ω) is calculated from

$$|\tilde{f}_g(t,\omega)|^2 = \left|\int_{-\infty}^{\infty} f(\tau)g(\tau - t)e^{-i\omega\tau}d\tau\right|^2. \tag{14.4.5}$$

(3) The time–frequency spread around a Gábor window is computed from the variance, or second moment, so that

$$\sigma_t^2 = \int_{-\infty}^{\infty} (\tau - t)^2 |g_{t,\omega}(\tau)|^2 d\tau = \int_{-\infty}^{\infty} \tau^2 |g(\tau)|^2 d\tau \tag{14.4.6a}$$

$$\sigma_\omega^2 = \frac{1}{2\pi}\int_{-\infty}^{\infty} (\nu - \omega)^2 |\tilde{g}_{t,\omega}(\nu)|^2 d\nu = \frac{1}{2\pi}\int_{-\infty}^{\infty} \nu^2 |\tilde{g}(\nu)|^2 d\nu \tag{14.4.6b}$$

where $\sigma_t\sigma_\omega$ is independent of t and ω and is governed by the Heisenberg uncertainty principle.

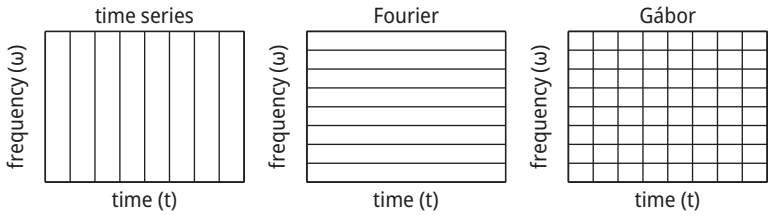


Figure 14.16 Graphical depiction of the difference between a time series analysis, Fourier analysis, and Gábor analysis of a signal. In the time series method, good resolution is achieved of the signal in the time domain, but no frequency resolution is achieved. In Fourier analysis, the frequency domain is well resolved at the expense of losing all time resolution. The Gábor method, or short-time Fourier transform, is constructed to give both time and frequency resolution. The area of each box can be constructed from $\sigma_t^2\sigma_\omega^2$.

(4) The Gábor transform is linear so that

$$\mathcal{G}[af_1 + bf_2] = a\mathcal{G}[f_1] + b\mathcal{G}[f_2]. \tag{14.4.7}$$

(5) The Gábor transform can be inverted with the formula

$$f(\tau) = \frac{1}{2\pi}\frac{1}{\|g\|^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\tilde{f}_g(t,\omega)g(\tau - t)e^{i\omega\tau}d\omega dt \tag{14.4.8}$$

where the integration must occur over all frequency- and time-shifting components. This double integral is in contrast to the Fourier transform, which requires only a single integration since it is a function, $\hat{f}(\omega)$, of the frequency alone.

Figure 14.16 is a cartoon representation of the fundamental ideas behind a time series analysis, Fourier transform analysis, and Gábor transform analysis of a given signal. In the time series method, good resolution is achieved of the signal in the time domain, but no frequency resolution is achieved. In Fourier analysis, the frequency domain is well resolved at the expense of losing all time resolution. The Gábor method, or short-time Fourier transform, trades away some measure of accuracy in both the time and frequency domains in order to give both time and frequency resolution simultaneously. Understanding this figure is critical to understanding the basic, high-level notions of time–frequency analysis.

In practice, the Gábor transform is computed by discretizing the time and frequency domains. Thus, a discrete version of the transform (14.3.2) needs to be considered. Essentially, by discretizing, the transform is done on a lattice of time and frequency. Thus, consider the lattice, or sample points,

$$\nu = m\omega_0 \tag{14.4.9a}$$

$$\tau = nt_0 \tag{14.4.9b}$$

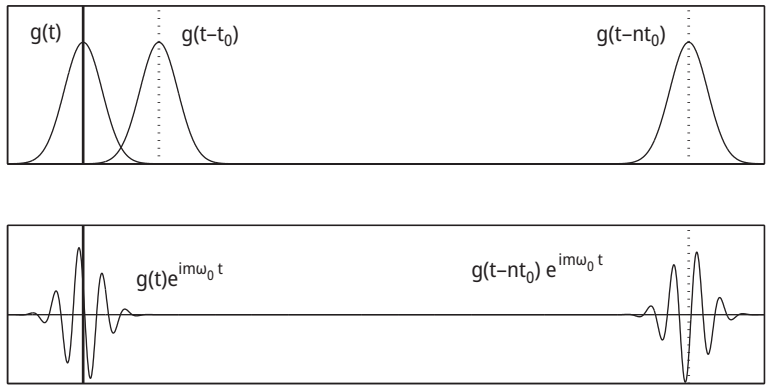


Figure 14.17 Illustration of the discrete Gábor transform that occurs on the lattice sample points (14.3.9b). In the top figure, the translation with $\omega_0 = 0$ is depicted. The bottom figure depicts both translation in time and frequency. Note that the Gábor frames (windows) overlap so that good resolution of the signal can be achieved in both time and frequency since $0 < t_0, \omega_0 < 1$.

where m and n are integers and $\omega_0, t_0 > 0$ are constants. Then, the discrete version of $g_{t,\omega}$ becomes

$$g_{m,n}(t) = e^{i2\pi m\omega_0 t} g(t - nt_0) \tag{14.4.10}$$

and the Gábor transform becomes

$$\tilde{f}(m,n) = \int_{-\infty}^{\infty} f(t)\bar{g}_{m,n}(t)dt = (\, f, g_{m,n}). \tag{14.4.11}$$

Note that if $0 < t_0, \omega_0 < 1$, then the signal is oversampled and time frames exist, which yield excellent localization of the signal in both time and frequency. If $\omega_0, t_0 > 1$, the signal is undersampled and the Gábor lattice is incapable of reproducing the signal. Figure 14.17 shows the Gábor transform on the lattice given by (14.3.9b). The overlap of the Gábor window frames ensures that good resolution in time and frequency of a given signal can be achieved.

14.2.1 Drawbacks of the Gábor (STFT) transform

Although the Gábor transform gives a method whereby time and frequency can be simultaneously characterized, there are obvious limitations to the method. Specifically, the method is limited by the time filtering itself. Consider the illustration of the method in Fig. 14.15. The time window filters out the time behaviour of the signal in a window centred at τ with width a. Thus, when considering the spectral content of this window, any portion of the signal with a wavelength longer than the window is completely lost. Indeed, since the Heisenberg relationship must hold, the shorter the time-filtering window, the less information there is concerning the frequency content. In contrast, longer windows retain more

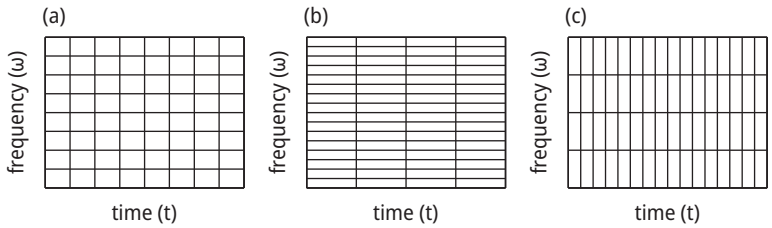


Figure 14.18 Illustration of the resolution trade-offs in the discrete Gábor transform. The figure (a) shows a time-filtering window that produces nearly equal localization of the time and frequency signal. By increasing the length of the filtering window, increased frequency resolution is gained at the expense of worse time resolution (b). Decreasing the time window does the opposite: time resolution is increased at the expense of poor frequency resolution (c).

frequency components, but this comes at the expense of losing the time resolution of the signal. Figure 14.18 provides a graphical description of the failings of the Gábor transform, specifically the trade-offs that occur between time and frequency resolution, and the fact that high accuracy in one of these comes at the expense of resolution in the other parameter. This is a consequence of a fixed time-filtering window.

14.2.2 Other short-time Fourier transform methods

The Gábor transform is not the only windowed Fourier transform that has been developed. There are several other well-used and highly developed STFT techniques. Here, a couple of these more highly used methods will be mentioned for completeness [51].

The *Zak transform* is closely related to the Gábor transform. It is also called the *Weil–Brezin* transform in harmonic analysis. First introduced by Gelfand in 1950 as a method for characterizing eigenfunction expansions in quantum mechanical systems with periodic potentials, it has been generalized to be a key mathematical tool for the analysis of Gábor transform methods. The Zak transform is defined as

$$\mathcal{L}_a f(t,\omega) = \sqrt{a} \sum_{n=-\infty}^{\infty} f(at + an) e^{-i2\pi n\omega} \tag{14.4.12}$$

where $a > 0$ is a constant and n is an integer. Two useful and key properties of this transform are as follows: $\mathcal{L}f(t,\omega+1) = \mathcal{L}f(t,\omega)$ (periodicity) and $\mathcal{L}f(t+1,\omega) = \exp(i2\pi\omega)\mathcal{L}f(t,\omega)$ (quasi-periodicity). These properties are particularly important for considering physical problems placed on a lattice.

The *Wigner–Ville distribution* is a particularly important transform in the development of radar and sonar technologies. Its various mathematical properties make it ideal for these applications and provide a method for achieving great time and frequency localization. The Wigner–Ville transform is defined as

$$\mathcal{W}_{f,g}(t,\omega) = \int_{-\infty}^{\infty} f(t+\tau/2)\bar{g}(t-\tau/2)e^{-i\omega}\tau d\tau \tag{14.4.13}$$

where this is a standard Fourier kernel that transforms the function $f(t+\tau/2)\bar{g}(t-\tau/2)$. This transform is nonlinear since $\mathcal{W}_{f_1+f_2,g_1+g_2} = \mathcal{W}_{f_1,g_1} + \mathcal{W}_{f_1,g_2} + \mathcal{W}_{f_2,g_1} + \mathcal{W}_{f_2,g_2}$ and $\mathcal{W}_{f+g} = \mathcal{W}_f + \mathcal{W}_g + 2\Re\{\mathcal{W}_{f,g}\}$.

Ultimately, alternative forms of the STFT are developed for one specific reason: to take advantage of some underlying properties of a given system. It is rare that a method developed for radar would be broadly applicable to other physical systems unless it were operating under the same physical principles. Regardless, one can see that speciality techniques exist for time–frequency analysis of different systems.

14.5 Time–Frequency Analysis and Wavelets

The Gábor transform established two key principles for joint time–frequency analysis: *translation* of a short-time window and *scaling* of the short-time window to capture finer time resolution. Figure 14.15 shows the basic concept introduced in the theory of windowed Fourier transforms. Two parameters are introduced to handle the *translation* and *scaling*, namely τ and a. The shortcoming of this method is that it trades off accuracy in time (frequency) for accuracy in frequency (time). Thus, the fixed window size imposes a fundamental limitation on the level of time–frequency resolution that can be obtained.

A simple modification to the Gábor method is to allow the scaling window (a) to vary in order to successively extract improvements in the time resolution. In other words, first the low-frequency (poor time resolution) components are extracted using a broad scaling window. The scaling window is subsequently shortened in order to extract out higher frequencies and better time resolution. By keeping a catalogue of the extracting process, both excellent time and frequency resolution of a given signal can be obtained. This is the fundamental principle of *wavelet theory*. The term wavelet means little wave and originates from the fact that the scaling window extracts out smaller and smaller pieces of waves from the larger signal.

Wavelet analysis begins with the consideration of a function known as the *mother wavelet*:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right) \tag{14.4.1}$$

where $a \neq 0$ and b are real constants. The parameter a is the scaling parameter illustrated in Fig. 14.15, whereas the parameter b now denotes the translation parameter (previously denoted by τ in Fig. 14.15). Unlike Fourier analysis, and very much like Gábor transforms, there is a vast variety of mother wavelets that can be constructed. In principle, the mother wavelet is designed to have certain properties that are somehow beneficial for a given problem. Thus, depending upon the application, different mother wavelets may be selected.

Ultimately, the wavelet is simply another expansion basis for representing a given signal or function. Thus, it is not unlike the Fourier transform that represents the signal as a series of sines and cosines. Historically, the first wavelet was constructed by Haar in 1910 [52]. Thus, the concepts and ideas of wavelets are a century old. However, their widespread use

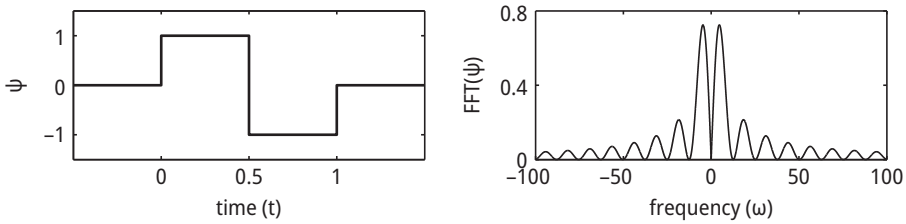


Figure 14.19 Representation of the compactly supported Haar wavelet function $\psi(t)$ and its Fourier transform $\hat{\psi}(\omega)$. Although highly localized in time due to the compact support, it is poorly localized in frequency with a decay of $1/\omega$.

and application did not become prevalent until the mid-1980s. The Haar wavelet is given by the piecewise constant function

$$\psi(t) = \begin{cases} 1 & 0 \le t < 1/2 \\ -1 & 1/2 \le t < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{14.4.2}$$

Figure 14.19 shows the Haar wavelet step function and its Fourier transform, which is a sinc-like function. Note further that $\int_{-\infty}^{\infty} \psi(t)dt = 0$ and $\|\psi(t)\|^2 = \int_{-\infty}^{\infty} |\psi(t)|^2 dt = 1$. The Haar wavelet is an ideal wavelet for describing localized signals in time (or space) since it has compact support. Indeed, for highly localized signals, it is much more efficient to use the Haar wavelet basis than the standard Fourier expansion. However, the Haar wavelet has poor localization properties in the frequency domain since it decays like a sinc function in powers of $1/\omega$. This is a consequence of the Heisenberg uncertainty principle.

To represent a signal with the Haar wavelet basis, the translation and scaling operations associated with the mother wavelet need to be considered. Depicted in Fig. 14.19 and given by (14.4.2) is the wavelet $\psi_{1,0}(t)$. Thus, its translation is zero and its scaling is unity. The concept in reconstructing a signal using the Haar wavelet basis is to consider decomposing the signal into more generic $\psi_{m,n}(t)$. By appropriate selection of the m and n, finer scales and appropriate locations of the signal can be extracted. For $a < 1$, the wavelet is a compressed version of $\psi_{1,0}$, whereas for $a > 1$, the wavelet is a dilated version of $\psi_{1,0}$. The scaling parameter a is typically taken to be a power of 2 so that $a = 2^j$ for some integer j. Figure 14.20 shows the compressed and dilated Haar wavelet for $a = 0.5$ and $a = 2$, i.e. $\psi_{1/2,0}$ and $\psi_{2,0}$. The compressed wavelet allows for finer scale resolution of a given signal, while the dilated wavelet captures low-frequency components of a signal by having a broad range in time.

The simple Haar wavelet already illustrates all the fundamental principles of the wavelet concept. Specifically by using scaling and translation, a given signal or function can be represented by a basis of functions, which allows for higher and higher refinement in the time resolution of a signal. Thus, it is much like the Gábor concept, except that now the time window is variable in order to capture different levels of resolution. Thus, the large-scale structures in time are captured with broad time-domain Haar wavelets. At this scale, the time resolution of the signal is very poor. However, by successive rescaling in time, a finer

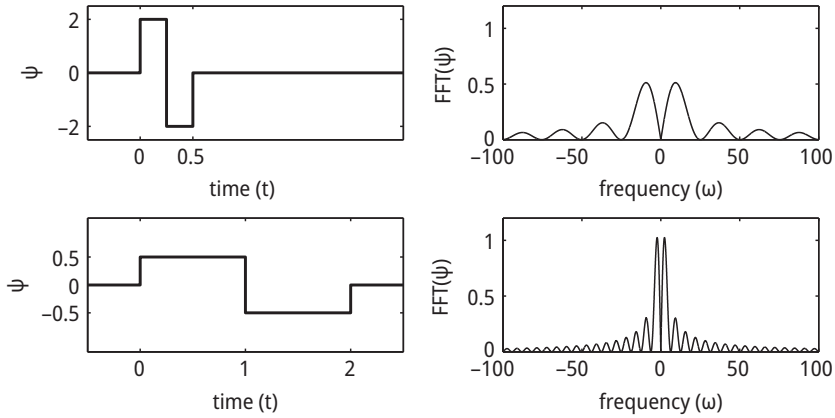


Figure 14.20 Illustration of the compression and dilation process of the Haar wavelet and its Fourier transform. In the top row, the compressed wavelet $\psi_{1/2,0}$ is shown. Improved time resolution is obtained at the expense of a broader frequency signature. The bottom row, shows the dilated wavelet $\psi_{2,0}$, which allows it to capture lower-frequency components of a signal.

and finer time resolution of the signal can be obtained along with its high-frequency components. The information at the low and high scales is all preserved so that a complete picture of the time–frequency domain can be constructed. Ultimately, the only limit in this process is the number of scaling levels to be considered.

The wavelet basis can be accessed via an integral transform of the form

$$(Tf)(\omega) = \int_t K(t,\omega) f(t) dt \tag{14.4.3}$$

where $K(t,\omega)$ is the kernel of the transform. This is equivalent in principle to the Fourier transform whose kernel is the oscillations given by $K(t,\omega) = \exp(-i\omega t)$. The key idea now is to define a transform that incorporates the mother wavelet as the kernel. Thus, we define the *continuous wavelet transform (CWT)*:

$$\mathcal{W}_\psi[f](a,b) = (f, \psi_{a,b}) = \int_{-\infty}^{\infty} f(t)\bar{\psi}_{a,b}(t) dt. \tag{14.4.4}$$

Much like the windowed Fourier transform, the CWT is a function of the dilation parameter a and translation parameter b. Parenthetically, a wavelet is admissible if the following property holds:

$$C_\psi = \int_{-\infty}^{\infty} \frac{\left|\hat{\psi}(\omega)\right|^2}{|\omega|} d\omega < \infty \tag{14.4.5}$$

where the Fourier transform of the wavelet is defined by

$$\hat{\psi}_{a,b} = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} e^{-i\omega t} \psi\left(\frac{t-b}{a}\right) dt = \frac{1}{\sqrt{|a|}} e^{-ib\omega} \hat{\psi}(a\omega). \tag{14.4.6}$$

Thus, provided the admissibility condition (14.4.5) is satisfied, the wavelet transform can be well defined.

As an example of the admissibility condition, consider the Haar wavelet (14.4.2). Its Fourier transform can be easily computed in terms of the sinc-like function:

$$\hat{\psi}(\omega) = ie^{-i\omega/2}\frac{\sin^2(\omega/4)}{\omega/4}. \tag{14.4.7}$$

Thus, the admissibility constant can be computed to be

$$\int_{-\infty}^{\infty}\frac{\left|\hat{\psi}(\omega)\right|^2}{|\omega|}d\omega = 16\int_{-\infty}^{\infty}\frac{1}{|\omega|^3}\left|\sin\frac{\omega}{4}\right|^4 d\omega < \infty. \tag{14.4.8}$$

This then shows that the Haar wavelet is in the admissible class.

Another interesting property of the wavelet transform is the ability to construct new wavelet bases. The following theorem is of particular importance.

Theorem: *If ψ is a wavelet and ϕ is a bounded integrable function, then the convolution $\psi \star \phi$ is a wavelet.*

In fact, from the Haar wavelet (14.4.2), we can construct new wavelet functions by convolving with for instance

$$\phi(t) = \begin{cases} 0 & t < 0 \\ 1 & 0 \le t \le 1 \\ 0 & t \ge 1 \end{cases} \tag{14.4.9}$$

or the function

$$\phi(t) = e^{-t^2}. \tag{14.4.10}$$

The convolutions of these functions ϕ with the Haar wavelet ψ (14.4.2) are produced in Fig. 14.21. These convolutions could also be used as mother wavelets in constructing a decomposition of a given signal or function.

The wavelet transform principle is quite simple. First, the signal is split up into a bunch of smaller signals by translating the wavelet with the parameter b over the entire time

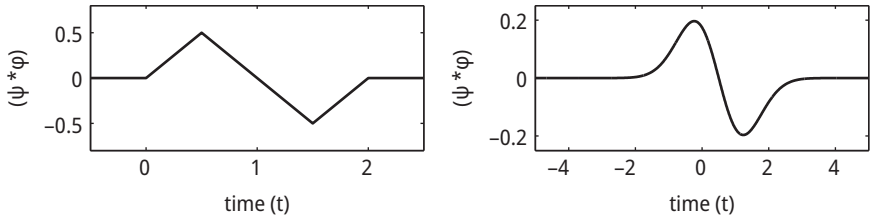


Figure 14.21 Convolution of the Haar wavelet with the functions (14.4.9) (left panel) and (14.4.10) (right panel). The convolved functions can be used as the mother wavelet for a wavelet basis expansion.

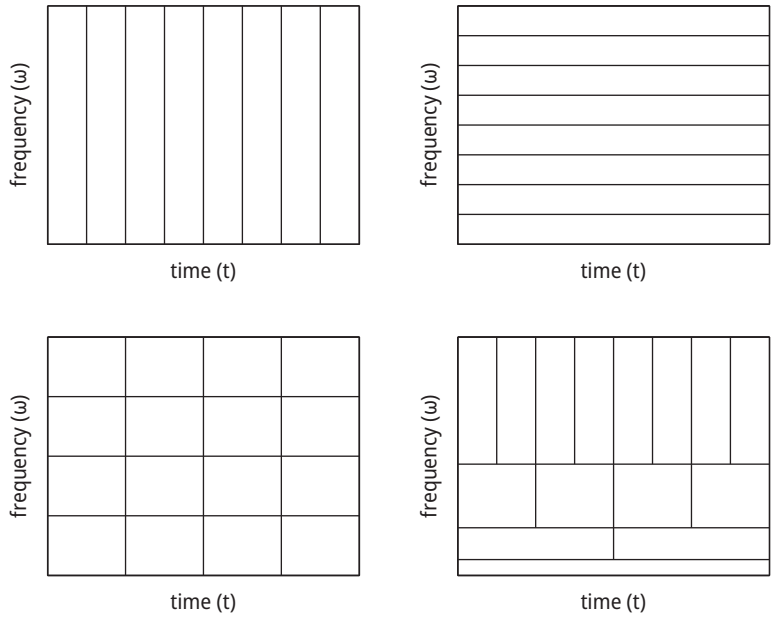


Figure 14.22 Graphical depiction of the difference between a time series analysis, Fourier analysis, Gábor analysis, and wavelet analysis of a signal. This figure is identical to Fig. 14.16 but with the inclusion of the time–frequency resolution achieved with the wavelet transform. The wavelet transform starts with a large Fourier domain window so that the entire frequency content is extracted. The time window is then scaled in half, leading to finer time resolution at the expense of worse frequency resolution. This process is continued until a desired time–frequency resolution is obtained.

domain of the signal. Second, the same signal is processed at different frequency bands, or resolutions, by scaling the wavelet window with the parameter a. The combination of translation and scaling allows for processing of the signals at different times and frequencies. Figure 14.22 is an upgrade of Fig. 14.16 that incorporates the wavelet transform concept in the time–frequency domain. In this figure, the standard time series, Fourier transform, and windowed Fourier transform are represented along with the multi-resolution concept of the wavelet transform. In particular, the box illustrating the wavelet transform shows the multi-resolution concept in action. Starting with a large Fourier domain window, the entire frequency content is extracted. The time window is then scaled in half, leading to finer time resolution at the expense of worse frequency resolution. This process is continued until a desired time–frequency resolution is obtained. This simple cartoon is critical for understanding wavelet application to time–frequency analysis.

14.5.1 Example: The Mexican hat wavelet

One of the more common wavelets is the Mexican hat wavelet. This wavelet is essentially a second moment of a Gaussian in the frequency domain. The definition of this wavelet and its transform are as follows:

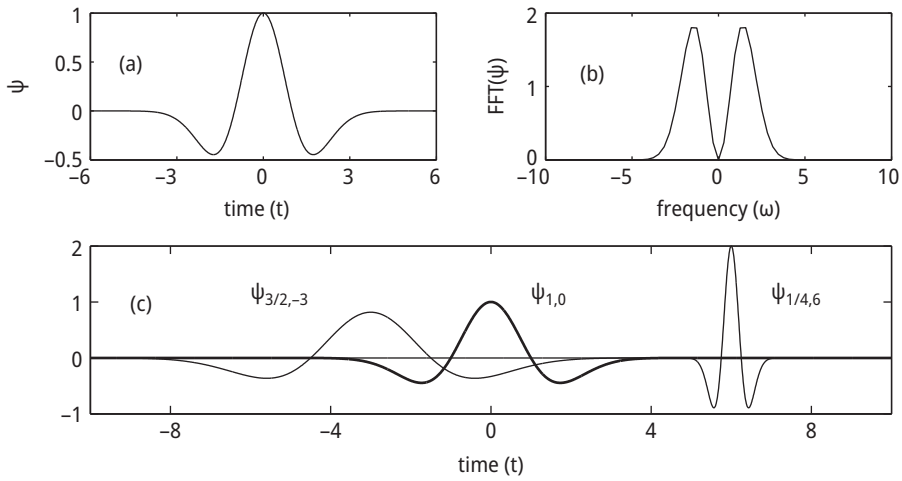


Figure 14.23 Illustration of the Mexican hat wavelet $\psi_{1,0}$ (top left panel), its Fourier transform $\hat{\psi}_{1,0}$ (top right panel), and two additional dilations and translations of the basic $\psi_{1,0}$ wavelet. Namely the $\psi_{3/2,-3}$ and $\psi_{1/4,6}$ are shown (bottom panel).

$$\psi(t) = (1-t^2)e^{-t^2/2} = -d^2/dt^2\left(e^{-t^2/2}\right) = \psi_{1,0} \tag{14.4.11a}$$

$$\hat{\psi}(\omega) = \hat{\psi}_{1,0}(\omega) = \sqrt{2\pi}\omega^2 e^{-\omega^2/2}. \tag{14.4.11b}$$

The Mexican hat wavelet has excellent localization properties in both time and frequency due to the minimal time–bandwidth product of the Gaussian function. Figure 14.23 (top panels) shows the basic Mexican wavelet function $\psi_{1,0}$ and its Fourier transform, both of which decay in t (ω) like $\exp(-t^2)$ $(\exp(-\omega^2))$. The Mexican hat wavelet can be dilated and translated easily as is depicted in Fig. 14.23 (bottom panel). Here three wavelets are depicted: $\psi_{1,0}$, $\psi_{3/2,-3}$, and $\psi_{1/4,6}$. This shows both the scaling and translation properties associated with any wavelet function.

To finish the initial discussion of wavelets, some of the various properties of the wavelets are listed. To begin, consider the time–frequency resolution and its localization around a given time and frequency. These quantities can be calculated from the relations:

$$\sigma_t^2 = \int_{-\infty}^{\infty} (t - \langle t \rangle)^2 \, |\psi(t)|^2 \, dt \tag{14.4.12a}$$

$$\sigma_\omega^2 = \frac{1}{2\pi} \int_{-\infty}^{\infty} (\omega - \langle \omega \rangle)^2 \left|\hat{\psi}(\omega)\right|^2 d\omega \tag{14.4.12b}$$

where the variances measure the spread of the time and frequency signal around $\langle t \rangle$ and $\langle \omega \rangle$, respectively. The Heisenberg uncertainty constrains the localization of time and frequency by the relation $\sigma_t^2 \sigma_\omega^2 \geq 1/2$. In addition, the CWT has the following mathematical properties:

(1) **Linearity** The transform is linear so that

$$\mathcal{W}_\psi(\alpha f + \beta g)(a,b) = \alpha \mathcal{W}_\psi(f)(a,b) + \beta \mathcal{W}_\psi(g)(a,b).$$

(2) **Translation** The transform has the translation property

$$\mathcal{W}_\psi(T_c f)(a,b) = \mathcal{W}_\psi(f)(a, b-c)$$

where $T_c f(t) = f(t-c)$.

(3) **Dilation** The dilation property follows

$$\mathcal{W}_\psi(D_c f)(a,b) = \frac{1}{\sqrt{c}} \mathcal{W}_\psi(f)(a/c, b/c)$$

where $c > 0$ and $D_c f(t) = (1/c) f(t/c)$.

(4) **Inversion** The transform can be inverted with the definition

$$f(t) = \frac{1}{C_\psi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathcal{W}_\psi(f)(a,b) \psi_{a,b}(t) \frac{dbda}{a^2}$$

where it becomes clear why the admissibility condition $C_\psi < \infty$ is needed.

To conclude this section, consider the idea of discretizing the wavelet transform on a computational grid. Thus, the transform is defined on a lattice so that

$$\psi_{m,n}(x) = a_0^{-m/2} \psi\left(a_0^{-m} x - n b_0\right) \tag{14.4.13}$$

where $a_0, b_0 > 0$ and m, n are integers. The *discrete wavelet transform* is then defined by

$$\begin{aligned} \mathcal{W}_\psi(f)(m,n) &= (f, \psi_{m,n}) \\ &= \int_{-\infty}^{\infty} f(t) \bar{\psi}_{m,n}(t) dt \\ &= a_0^{-m/2} \int_{-\infty}^{\infty} f(t) \bar{\psi}(a_0^{-m} t - n b_0) dt. \end{aligned} \tag{14.4.14}$$

Furthermore, if $\psi_{m,n}$ are complete, then a given signal or function can be expanded in the wavelet basis:

$$f(t) = \sum_{m,n=-\infty}^{\infty} (f, \psi_{m,n}) \psi_{m,n}(t). \tag{14.4.15}$$

This expansion is in a set of wavelet frames. It still needs to be determined if the expansion is in terms of a set of basis functions. It should be noted that the scaling and dilation parameters are typically taken to be $a_0 = 2$ and $b_0 = 1$, corresponding to dilations of 2^{-m} and translations of $n2^m$. Figure 14.24 gives a graphical depiction of the time–frequency discretization of the wavelet transform. This figure is especially relevant for the computational evaluation of the wavelet transform. Further, it is the basis of fast algorithms for multi-resolution analysis.

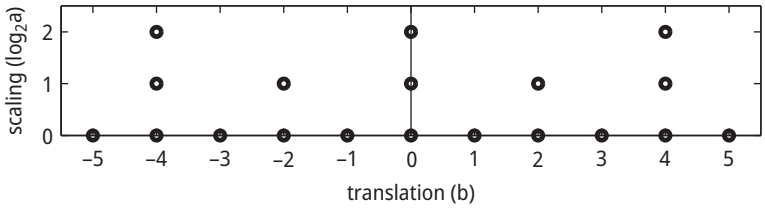


Figure 14.24 Discretization of the discrete wavelet transform with $a_0 = 2$ and $b_0 = 1$. This figure is a more formal depiction of the multi-resolution discretization as shown in Fig. 14.22.

14.6 Multi-Resolution Analysis and the Wavelet Basis

Before proceeding forward with wavelets, it is important to establish some key mathematical properties. Indeed, the most important issue to resolve is the ability of the wavelet to actually represent a given signal or function. In Fourier analysis, it has been established that any generic function can be represented by a series of cosines and sines. Something similar is needed for wavelets in order to make them a useful tool for the analysis of time–frequency signals.

The concept of a wavelet is simple and intuitive: construct a signal using a single function $\psi \in L^2$, which can be written as $\psi_{m,n}$ and that represents binary dilations by 2^m and translations of $n2^{-m}$ so that

$$\psi_{m,n} = 2^{m/2}\psi\left(2^m\left(x - n/2^m\right)\right) = 2^{m/2}\psi(2^m x - n) \tag{14.5.1}$$

where m and n are integers. The use of this wavelet for representing a given signal or function is simple enough. However, there is a critical issue to be resolved concerning the *orthogonality* of the functions $\psi_{m,n}$. Ultimately, this is the primary issue that must be addressed in order to consider the wavelets as basis functions in an expansion. Thus, we define the orthogonality condition as

$$(\psi_{m,n}, \psi_{k,l}) = \int_{-\infty}^{\infty} \psi_{m,n}(x)\psi_{k,l}(x)dx = \delta_{m,k}\delta_{n,l} \tag{14.5.2}$$

where δ_{ij} is the Dirac delta defined by

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{14.5.3}$$

where i, j are integers. This statement of orthogonality is generic, and it holds in most function spaces with a defined inner product.

The importance of orthogonality should not be underestimated. It is very important in applications where a functional expansion is used to approximate a given function or solution. In what follows, two examples are given concerning the key role of orthogonality.

14.6.1 Fourier expansions

Consider representing an even function $f(x)$ over the domain $x \in [0, L]$ with a cosine expansion basis. By Fourier theory, the function can be represented by

$$f(x) = \sum_{n=0}^{\infty} a_n \cos \frac{n\pi x}{L} \tag{14.5.4}$$

where the coefficients a_n can be constructed by using inner product rules and orthogonality. Specifically, by multiplying both sides of the equation by $\cos(m\pi x/L)$ and integrating over $x \in [0, L]$, i.e. taking the inner product with respect to $\cos(m\pi x/L)$, the following result is found:

$$(\, f, \cos m\pi x/L) = \sum_{n=0}^{\infty} a_n \, (\cos n\pi x/L, \cos m\pi x/L). \tag{14.5.5}$$

This is where orthogonality plays a key role: the infinite sum on the right-hand side can be reduced to a single index, where $n = m$ since the cosines are orthogonal to each other

$$(\cos n\pi x/L, \cos m\pi x/L) = \begin{cases} 0 & n \neq m \\ L & n = m. \end{cases} \tag{14.5.6}$$

Thus, the coefficients can be computed to be

$$a_n = \frac{1}{L} \int_0^L f(x) \cos \frac{n\pi x}{L} dx \tag{14.5.7}$$

and the expansion is accomplished. Moreover, the cosine basis is complete for even functions, and any signal or function $f(x)$ can be represented, i.e. as $n \to \infty$ in the sum, the expansion converges to the given signal $f(x)$.

14.6.2 Eigenfunction expansions

The cosine expansion is a subset of the more general eigenfunction expansion technique that is often used to solve differential and partial differential equation problems. Consider the nonhomogeneous boundary value problem

$$Lu = f(x) \tag{14.5.8}$$

where L is a given self-adjoint linear operator. This problem can be solved with an eigenfunction expansion technique by considering the associated eigenvalue problem of the operator L:

$$Lu_n = \lambda_n u_n. \tag{14.5.9}$$

The solution of (14.5.11) can then be expressed as

$$u(x) = \sum_{n=0}^{\infty} a_n u_n \tag{14.5.10}$$

provided the coefficients a_n can be determined. Plugging in this solution to (14.5.11) yields the following calculations:

$$\begin{aligned} Lu &= f \\ L\left(\sum a_n u_n\right) &= f \\ \sum a_n L u_n &= f \\ \sum a_n \lambda_n u_n &= f. \end{aligned} \tag{14.5.11}$$

Taking the inner product of both sides with respect to u_m yields

$$\begin{aligned} \left(\sum a_n \lambda_n u_n, u_m\right) &= (f, u_m) \\ \sum a_n \lambda_n (u_n, u_m) &= (f, u_m) \\ a_m \lambda_m &= (f, u_m) \end{aligned} \tag{14.5.12}$$

where the last line is achieved by orthogonality of the eigenfunctions $(u_n, u_m) = \delta_{n,m}$. This then gives $a_m = (f, u_m)/\lambda_m$ and the eigenfunction expansion solution is

$$u(x) = \sum_{n=0}^{\infty} \frac{(f, u_n)}{\lambda_n} u_n. \tag{14.5.13}$$

Provided the u_n are a complete set, this expansion is guaranteed to converge to $u(x)$ as $n \to \infty$.

14.6.3 Orthonormal wavelets

The preceding examples highlight the importance of orthogonality for representing a given function. A wavelet ψ is called orthogonal if the family of functions $\psi_{m,n}$ are orthogonal as given by (14.5.2). In this case, a given signal or function can be uniquely expressed with the doubly infinite series

$$f(t) = \sum_{n,m=-\infty}^{\infty} c_{m,n} \psi_{m,n}(t) \tag{14.5.14}$$

where the coefficients are given from orthogonality by

$$c_{m,n} = (f, \psi_{m,n}). \tag{14.5.15}$$

The series is guaranteed to converge to $f(t)$ in the L^2 norm.

The above result based upon orthogonal wavelets establishes the key mathematical framework needed for using wavelets in a very broad and general way. It is this result that allows us to think of wavelets philosophically as the same as the Fourier transform.

14.6.4 Multi-resolution analysis (MRA)

The power of the wavelet basis is its ability to take a function or signal $f(t)$ and express it as a limit of successive approximations, each of which is a finer and finer version of the function in time. These successive approximations correspond to different resolution levels.

A *multi-resolution analysis*, commonly referred to as an MRA, is a method that gives a formal approach to constructing the signal with different resolution levels. Mathematically, this involves a sequence

$$\{V_m : \ m \in \text{integers}\} \tag{14.5.16}$$

of embedded subspaces of L^2 that satisfies the following relations:

(1) The subspaces can be embedded in each other so that

$$Coarse \ \cdots \subset V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \cdots V_m \subset V_{m+1} \cdots \ Fine.$$

(2) The union of all the embedded subspaces spans the entire L^2 space so that

$$\cup_{m=-\infty}^{\infty} V_m$$

is dense in L^2.

(3) The intersection of subspaces is the null set so that

$$\cap_{m=-\infty}^{\infty} V_m = \{0\}.$$

(4) Each subspace picks up a given resolution so that $f(x) \in V_m$ if and only if $f(2x) \in V_{m+1}$ for all integers m.

(5) There exists a function $\phi \in V_0$ such that

$$\{\phi_{0,n} = \phi(x-n)\}$$

is an orthogonal basis for V_0 so that

$$\| f\|^2 = \int_{-\infty}^{\infty} |\, f(x)|^2 \, dx = \sum_{-\infty}^{\infty} |(\, f, \phi_{0,n})|^2 \,.$$

The function ϕ is called the *scaling function* or *father wavelet*.

If $\{V_m\}$ is a multi-resolution of L^2 and if V_0 is the closed subspace generated by the integer translates of a single function ϕ, then we say ϕ generates the MRA.

One remark of importance: since $V_0 \subset V_1$ and ϕ is a scaling function for V_0 and also for V_1, then

$$\phi(x) = \sum_{-\infty}^{\infty} c_n \phi_{1,n}(x) = \sqrt{2} \sum_{-\infty}^{\infty} c_n \phi(2x-n) \tag{14.5.17}$$

where $c_n = (\phi, \phi_{1,n})$ and $\sum_{-\infty}^{\infty} |c_n|^2 = 1$. This equation, which relates the scaling function as a function of x and $2x$, is known as the *dilation equation*, or *two-scale equation*, or *refinement equation* because it reflects $\phi(x)$ in the refined space V_1 which has the finer scale of 2^{-1}.

Since $V_m \subset V_{m+1}$, we can define the orthogonal complement of V_m in V_{m+1} as

$$V_{m+1} = V_m \oplus W_m \tag{14.5.18}$$

where $V_m \perp W_m$. This can be generalized so that

$$\begin{aligned} V_{m+1} &= V_m \oplus W_m \\ &= (V_{m-1} \oplus W_{m-1}) \oplus W_m \\ &\vdots \\ &= V_0 \oplus W_0 \oplus W_1 \oplus \cdots \oplus W_m \\ &= V_0 \oplus (\oplus_{n=0}^{m} W_n). \end{aligned} \tag{14.5.19}$$

As $m \to \infty$, it can be found that

$$V_0 \oplus (\oplus_{n=0}^{\infty} W_n) = L^2. \tag{14.5.20}$$

In a similar fashion, the resolution can rescale upwards so that

$$\oplus_{n=-\infty}^{\infty} W_n = L^2. \tag{14.5.21}$$

Moreover, there exists a scaling function $\psi \in W_0$ (the mother wavelet) such that

$$\psi_{0,n}(x) = \psi(x - n) \tag{14.5.22}$$

constitutes an orthogonal basis for W_0 and

$$\psi_{m,n}(x) = 2^{m/2} \psi(2^m x - n) \tag{14.5.23}$$

is an orthogonal basis for W_m. Thus, the mother wavelet ψ spans the orthogonal complement subset W_m, while the scaling function ϕ spans the subsets V_m. The connection between the father and mother wavelet is shown in the following theorem.

Theorem: *If $\{V_m\}$ is a MRA with scaling function ϕ, then there is a mother wavelet ψ*

$$\psi(x) = \sqrt{2} \sum_{-\infty}^{\infty} (-1)^{n-1} \bar{c}_{-n-1} \phi(2x - n) \tag{14.5.24}$$

where

$$c_n = (\phi, \phi_{1,n}) = \sqrt{2} \int_{-\infty}^{\infty} \phi(x) \bar{\phi}(2x - 1) dx \,. \tag{14.5.25}$$

That is, the system $\psi_{m,n}(x)$ is an orthogonal basis of L^2.

This theorem is critical for what we would like to do. Namely, use the wavelet basis functions as a complete expansion basis for a given function $f(x)$ in L^2. Further, it explicitly states the connection between the *scaling function* $\phi(x)$ (father wavelet) and *wavelet function* $\psi(x)$ (mother wavelet). It is only left to construct a desirable wavelet basis to use. As for wavelet construction, the idea is to build them to take advantage of certain properties of the system so that it gives an efficient and meaningful representation of your time–frequency data.

14.7 Spectrograms and the Gábor Transform in Python

The aim of this section will be to use Python's fast Fourier transform routines modified to handle the Gábor transform. The Gábor transform allows for a fast and easy way to analyse both the time and frequency properties of a given signal. Indeed, this windowed Fourier transform method is used extensively for analysing speech and vocalization patterns. For such applications, it is typical to produce a *spectrogram* that represents the signal in both the time and frequency domain. Figures 14.25 and 14.26 are produced from the vocalization patterns in time–frequency of a human saying "do re mi" and a humpback whale vocalizing to other whales. The time–frequency analysis can be used to produce speech recognition algorithms given the characteristic signatures in the time–frequency domains of sounds. Thus, spectrograms are a sort of fingerprint of sound.

To understand the algorithms which produce the spectrogram, it is informative to return to the characteristic picture shown in Fig. 14.15. This demonstrates the action of an applied time filter in extracting time localization information. To build a specific example, consider

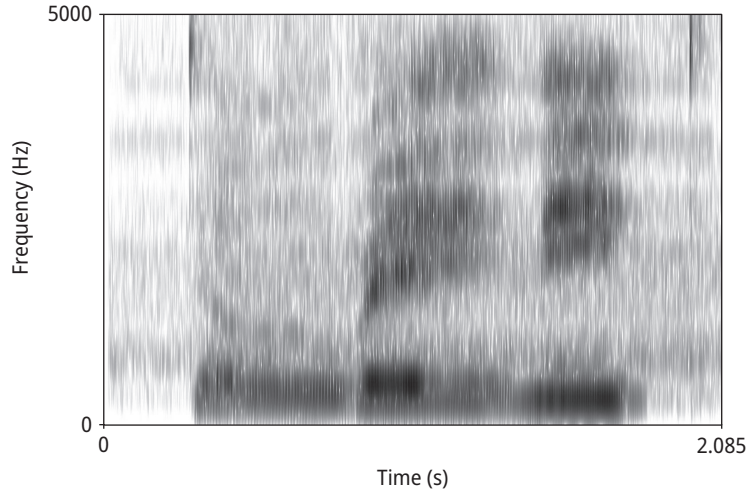


Figure 14.25 Spectrogram (time–frequency) analysis of a male human saying "do re mi." The spectrogram is created with the software program Praat, which is an open-source code for analysing phonetics.

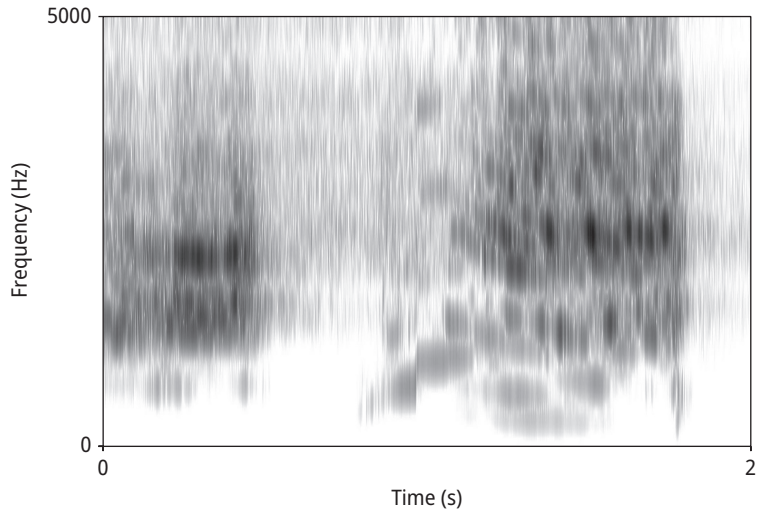


Figure 14.26 Spectrogram (time–frequency) analysis of a humpback whale vocalization over a short period of time. The spectrogram is created with the software program Praat, which is an open-source code for analysing phonetics.

the following Python code that builds a time domain (t), its corresponding Fourier domain (ω), a relatively complicated signal ($S(t)$), and its Fourier transform ($\hat{S}(\omega)$).

```
L = 10; n = np.power(2,11)
t = np.linspace(0, L, n+1); t = t[:-1]
k = (2*np.pi/(L))*np.concatenate((np.arange(0,n/2), np.arange(-n/2,0)))
ks = fftshift(k)

S = (3*np.sin(2*t) + 0.5*np.sinh(0.5*(t-3))/np.cosh(0.5*(t-3))
  + 0.2*np.exp(-(t-4)**2) + 1.5*np.sin(5*t) + 4*np.cos(3*(t-6)
     **2))/10 + (t/20)**3
St = fft(S)
```

The signal and its Fourier transform can be plotted. Figure 14.27 shows the signal and its Fourier transform for the above example. This signal $S(t)$ will be analysed using the Gábor transform method.

The simplest Gábor window to implement is a Gaussian time filter centred at some time τ with width a. As has been demonstrated, the parameter a is critical for determining the level of time resolution versus frequency resolution in a time–frequency plot. Figure 14.28 shows the signal under consideration with three filter widths. The narrower the time-filtering, the

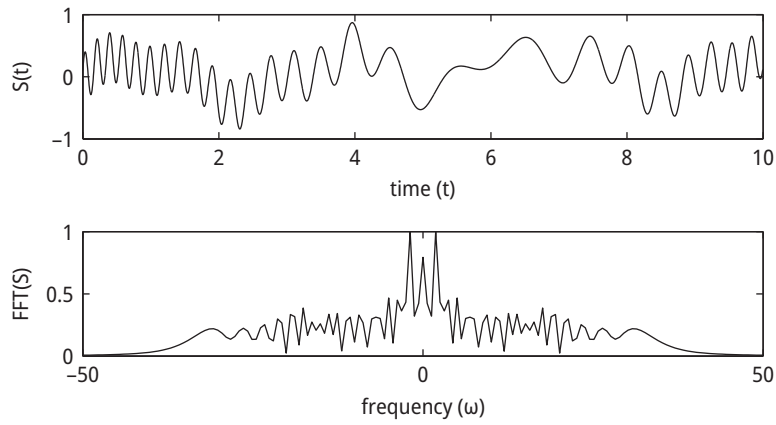


Figure 14.27 Time signal and its Fourier transform considered for a time–frequency analysis in what follows.

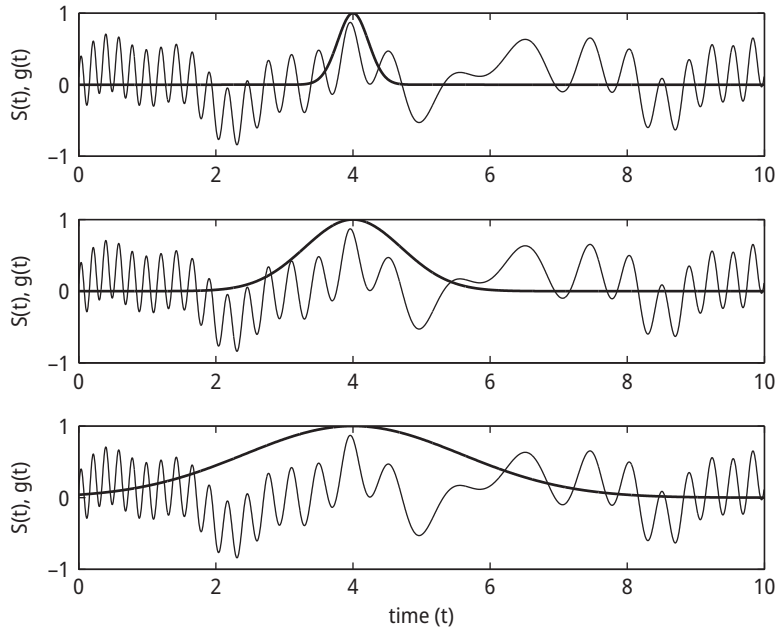


Figure 14.28 Time signal $S(t)$ and the Gábor time filter $g(t)$ (bold lines) for three different Gaussian filters: $g(t) = \exp(-10(x-4)^2)$ (top), $g(t) = \exp(-(x-4)^2)$ (middle), and $g(t) = \exp(-0.2(x-4)^2)$ (bottom). The different filter widths determine the time–frequency resolution. Better time resolution gives worse frequency resolution and vice-versa due to the Heisenberg uncertainty principle.

better resolution in time. However, this also produces the worst resolution in frequency. Conversely, a wide window in time produces much better frequency resolution at the expense of reducing the time resolution. A simple extension to the existing code produces a signal plot along with three different filter widths of Gaussian shape.

```
plt.rcParams['figure.figsize'] = [10, 10]
fig=plt.figure()

for j in np.arange(0,len(width)):
    g = np.exp(-width[j]*(t-4)**2)
    ax = fig.add_subplot(311+j)
    ax.plot(t,S,'r',t,g,'k')
    ax.set_ylabel('S, g')
    ax.set_xlabel('time')
```

The key now for the Gábor transform is to multiply the time filter Gábor function $g(t)$ with the original signal $S(t)$ in order to produce a windowed section of the signal. The Fourier transform of the windowed section then gives the local frequency content in time. The following code constructs the windowed Fourier transform with the Gábor filtering function:

$$g(t) = e^{-a(t-b)^2}. \tag{14.6.1}$$

The Gaussian filtering has a width parameter a and translation parameter b. The following code constructs the windowed Fourier transform using the Gaussian with $a = 2$ and $b = 4$.

```
g = np.exp(-2*(t-4)**2)
Sg=g*S
Sgt=fftshift(fft(Sg))
```

Figure 14.29 demonstrates the application of this code and the windowed Fourier transform in extracting local frequencies of a local time window.

The key to generating a spectrogram is to now vary the position b of the time filter and produce spectra at each location in time. In theory, the parameter b is continuously translated to produce the time–frequency picture. In practice, like everything else, the parameter b is discretized. The level of discretization is important in establishing a good time–frequency analysis. Specifically, finer resolution will produce better results. The following code makes a dynamical *movie* of this process as the parameter b is translated.

```
dt=0.1; tslide = np.arange(0,10+dt,dt)

for j in np.arange(0,len(tslide)):
    g = np.exp(-1*(t-tslide[j])**2)
    Sg=g*S
    Sgt=fftshift(fft(Sg))

    ax = fig.add_subplot(311)
    ax.plot(t,S,'r',t,g,'k')

    ax = fig.add_subplot(312)
    ax.plot(t,Sg,'k')

    ax = fig.add_subplot(313)
    ax.plot(ks,abs(Sgt),'k')
    ax.set_xlim(-100, 100)
```

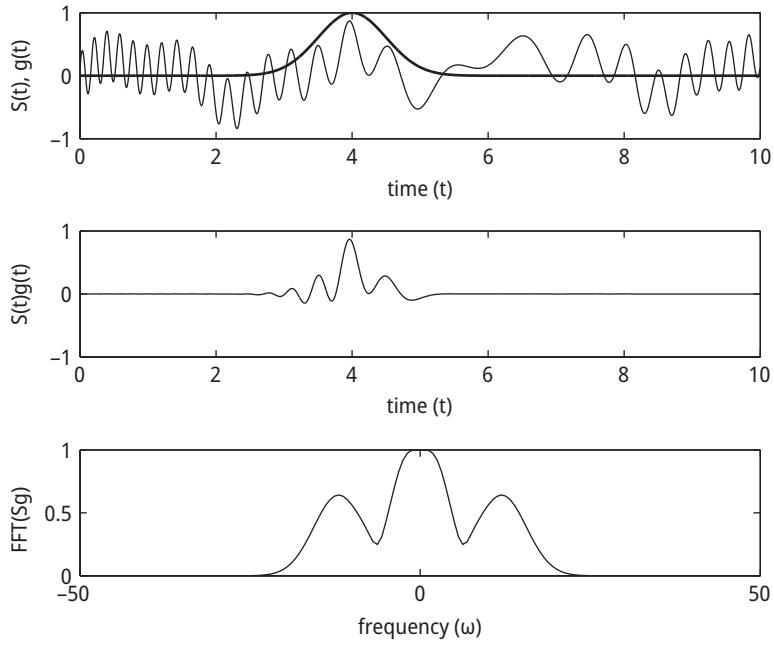


Figure 14.29 Time signal $S(t)$ and the Gábor time filter $g(t) = \exp(-2(x-4)^2)$ (bold line) for a Gaussian filter. The product $S(t)g(t)$ is depicted in the middle panel and its Fourier transform $\hat{Sg}(\omega)$ is depicted in the bottom panel. Note that the windowing of the Fourier transform can severely limit the detection of low-frequency components.

This movie is particularly illustrative and provides an excellent graphical representation of how the Gábor time-filtering extracts both local time information and local frequency content. It also illustrates, as the parameter a is adjusted, the ability (or inability) of the windowed Fourier transform to provide accurate time and frequency information.

The code just developed also produces a matrix **Sgt_spec**, which contains the Fourier transform at each slice in time of the parameter b. It is this matrix that produces the spectrogram of the time–frequency signal. The spectrogram can be viewed with the commands

```
plt.pcolor(tslide,ks,Sgt_spec.')
```

Modifying the code slightly, a spectrogram of the signal $S(t)$ can be made for three different filter widths $a = 5, 1, 0.2$ in (14.6.1). The spectrograms are shown in Fig. 14.30 where from left to right the filtering window is broadened from $a = 5$ to $a = 0.2$. Note that for the left plot, strong localization of the signal in time is achieved at the expense of suppressing almost all the low-frequency components of the signal. In contrast, the rightmost figure

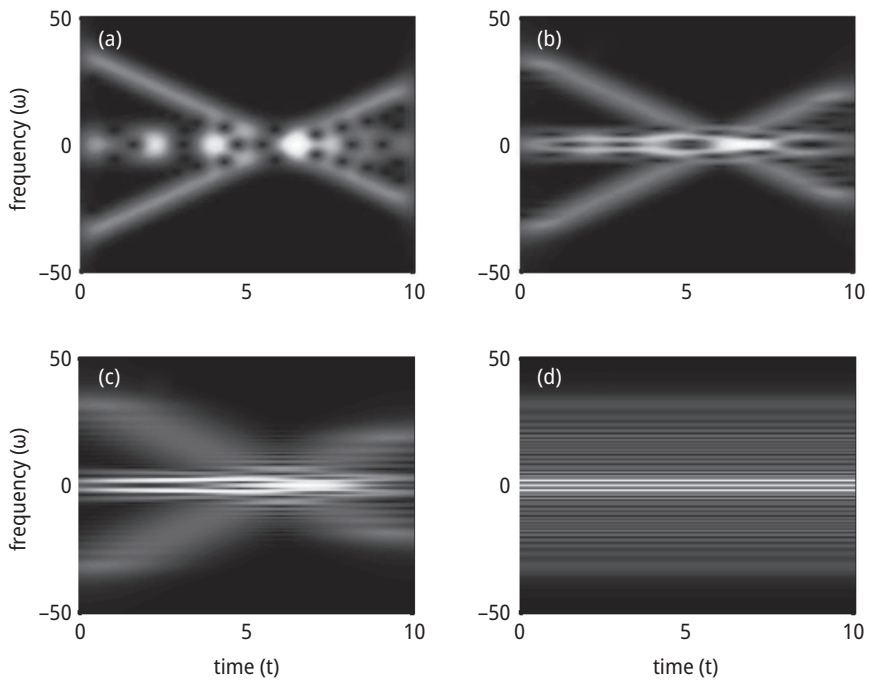


Figure 14.30 Spectrograms produced from the Gábor time-filtering Eq. (14.6.1) with (a) $a = 5$, (b) $a = 1$, and (c) $a = 0.2$. The Fourier transform, which has no time localization information is depicted in (d). It is easily seen that the window width trades off time and frequency resolution at the expense of each other. Regardless, the spectrogram gives a visual time–frequency picture of a given signal.

with a wide temporal filter preserves excellent resolution of the Fourier domain but fails to localize signals in time. Such are the trade-offs associated with a fixed Gábor window transform.

The wavelet toolbox also allows for considering a signal with a wavelet expansion. The continuous wavelet 1D provides a full decomposition of the signal into its wavelet basis. Figures 14.31 and 14.32 show a given signal along with the calculation of the wavelet coefficients $C_{a,b}$, where a and b are dilation and translation operations, respectively. This provides the basis for a time–frequency analysis. Additionally, given a chosen level of resolution determined by the parameter a, then the third panel shows the parameter $C_{a,b}$ for $a = a_{\max}/2$. The local maxima of the $C_{a,b}$ function are also shown. As before, different wavelet bases may be chosen along with different levels of resolution of a given signal.

The wavelets transform and its algorithms can be directly accessed from the command line, just like the filter design toolbox. For instance, to apply a continuous wavelet transform, the command

```
import pywt
coeffs, _ = pywt.cwt(S, scales, 'wname')
```

can be used to compute the real or complex continuous 1D wavelet coefficients. Specifically, it computes the continuous wavelet coefficients of the vector S at real, positive scales, using a wavelet whose name is wname (for instance, haar). The signal S is real, the wavelet can be real or complex.

14.7.1 JPEG compression with wavelets

Wavelets can be directly applied to the arena of image compression. To illustrate the above concepts more clearly, we can consider a digital image with 600×800 pixels (rows $\times$ columns). Figure 14.31(a) shows the original image. The following Python code imports the original JPEG picture and plots the results. It also constructs the matrix **Abw**, which is the picture in matrix form. The image is well defined and of high resolution. Indeed, it contains $600 \times 800 = 480\,000$ pixels encoding the image. However, the image can actually be encoded with far less data. Specifically, it has been well-known for the past two decades that wavelets are an ideal basis in which to encode photos.

To demonstrate the nearly optimal encoding of the wavelet basis, the original photo is subjected to a 2D wavelet transform:

```
coeffs, _ = pywt.wavedec2(Abw, 'db1', level=2)
```

The **wavedec** command performs a 2D transform and returns the wavelet coefficient vector $\mathbf{C}$ along with its ordering matrix $\mathbf{S}$. In this example, a two-level wavelet decomposition is

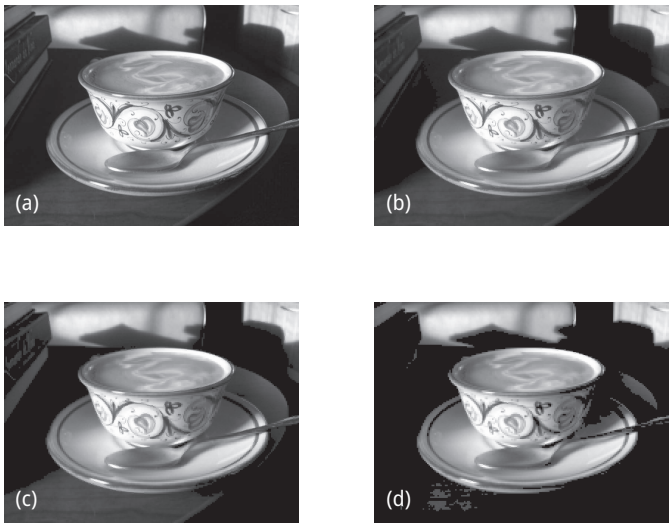


Figure 14.31 Original image (a) along with the reconstructed image using approximately (b) 5.6%, (c) 4.5%, or (d) 3% of the information encoded in the wavelet coefficients. At 3%, the image reconstruction starts to fail. This reconstruction process illustrates that images are sparse in the wavelet representation.

performed (thus the value of 2 in the **wavedec** command call) using Daubechies wavelets (thus the **db1** option). Higher-level wavelet decompositions can be performed using a variety of wavelets.

In the wavelet basis, the image is expressed as a collection of weightings of the wavelet coefficients. This is identical to representing a time-domain signal as a collection of frequencies using the Fourier transform. Figure 14.32(a) depicts the 480 000 wavelet coefficient values for the given picture. A zoom-in of the dense-looking region around 0 to 0.1 is shown in Fig. 14.33(a). In zooming in, one can clearly see that many of the coefficients that make up the image are nearly zero. Due to numerical round-off, for instance, none of the coefficients will actually be zero.

Image compression follows directly from this wavelet analysis. Specifically, one can simply choose a threshold and directly set all wavelet coefficients below this chosen threshold to be identically zero instead of nearly zero. What remains is to demonstrate that the image can be reconstructed using such a strategy. If it works, then one would only need to retain the much smaller (sparse) number of nonzero wavelets in order to store and/or reconstruct the image.

To address the image compression issue, we can apply a threshold rule to the wavelet coefficient vector $\mathbf{C}$ that is shown in Fig. 14.32(a). The following code successively sets a threshold at 50, 100, and 200 in order to generate a sparse wavelet coefficient vector $\mathbf{C2}$. This new sparse vector is then used to encode and reconstruct the image. The results of the compression process are illustrated in Figs. 14.31–14.33.

To understand the compression process, first consider Fig. 14.32, which shows the wavelet coefficient vector for the three different threshold values of (b) 50, (c) 100, and (d)

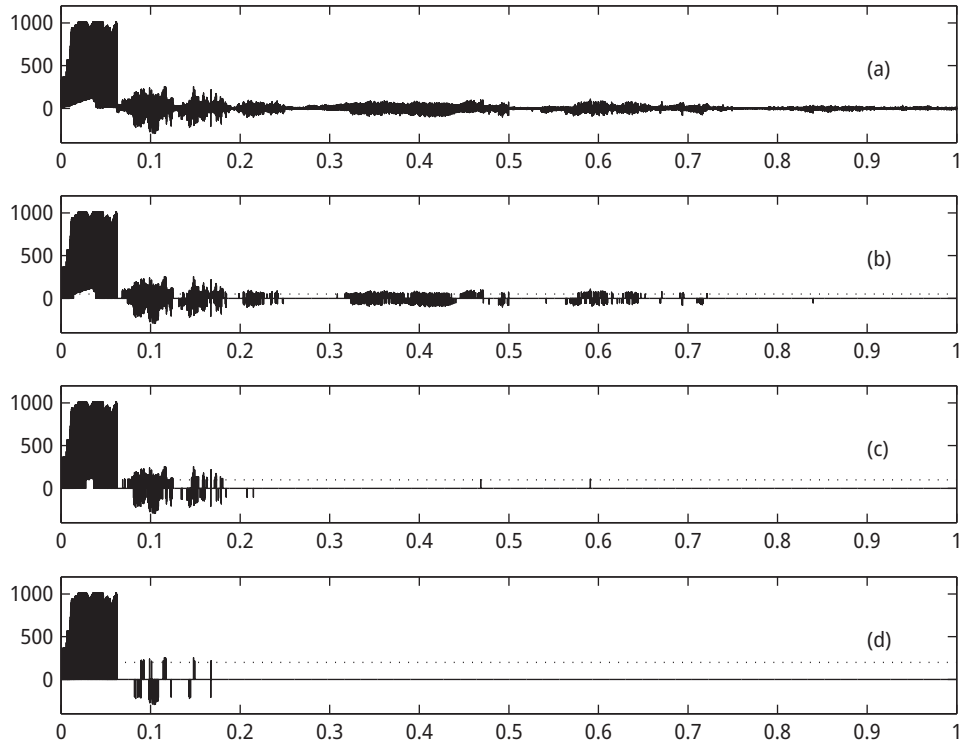


Figure 14.32 Original wavelet coefficient representation of the image (a) along with the thresholded coefficient vector for a threshold (dotted line) value of (b) 50, (c) 100, and (d) 200. The percentage of identically zero wavelet coefficients are 94.4%, 95.5%, and 97.0%, respectively. Only the non-zero coefficients are kept for encoding of the image. The x-axis has been normalized to unity.

200. The threshold line for each of these is shown as the dotted line. Anything below the threshold line is set identically to zero; thus, it can be discarded when encoding the image. Figure 14.33 shows a zoom-in of the region near $0 \text{ to } 0.1$. In addition to setting a threshold, the above code also calculates the percentage of zero wavelet coefficients. For the three threshold values considered here, the percentage of wavelet coefficients that are zero are 94.4%, 95.5%, and 97.0%, respectively. Thus, only approximately 5.6%, 4.5%, or 3% of the information is needed for the image reconstruction. Indeed, this is the key idea behind the JPEG2000 image compression algorithm: you only save what is needed to reconstruct the image to a desired level of accuracy.

The actual image reconstruction is illustrated in Fig. 14.31 for the three different levels of thresholding. From (b) to (d), the image is reconstructed from 5.6%, 4.5%, or 3% of the original information. At 3%, the image finally becomes visibly worse and begins to fail in faithfully reproducing the original image. At 5.6% or 4.5%, it is difficult to tell much difference with the original image. This is quite remarkable and illustrates the *sparsity* of the image in the wavelet basis. Such sparsity is ubiquitous for image representation using wavelets.

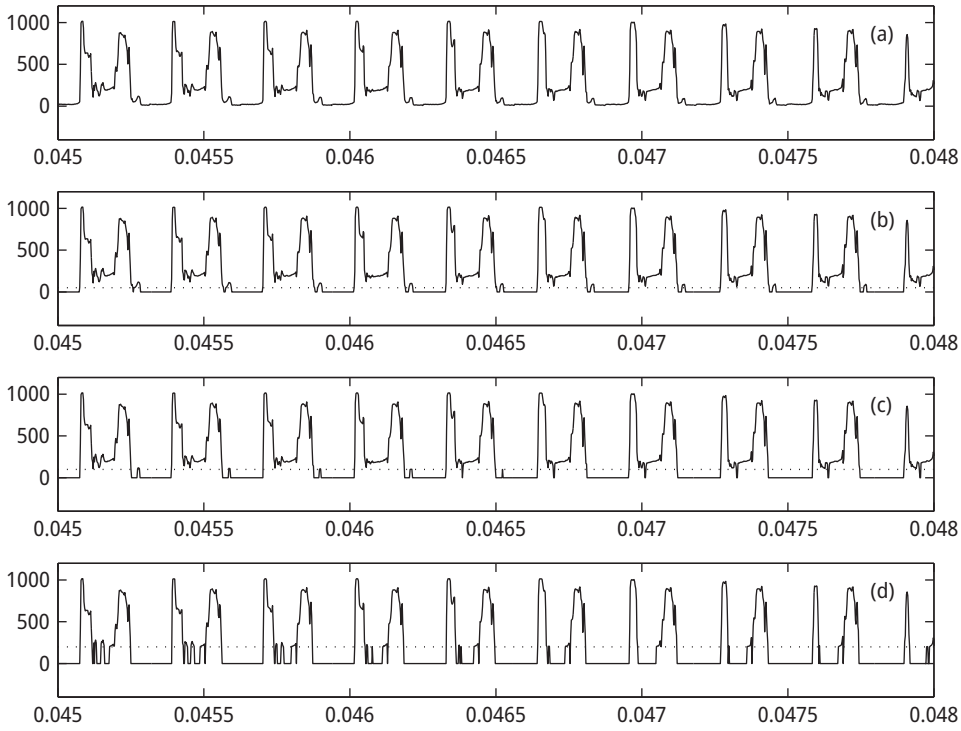


Figure 14.33 Zoom-in of the original wavelet coefficient representation of the image (a) along with the thresholded coefficient vector for a threshold (dotted line) value of (b) 50, (c) 100, and (d) 200 (See Fig. 14.32). In the zoomed-in region, one can clearly see that many of the coefficients are below threshold. The zoom-in is over the entire wavelet domain that has been normalized to unity.

14.8 **Image Processing and Denoising**

Over the past couple of decades, imaging science has had a profound impact on science, engineering, medicine, and technology. The reach of imaging science is immense, spanning the range of modern day biological imaging tools to computer graphics. But the imaging methods are not limited to visual data. Instead, a general mathematical framework has been developed by which more general data analysis can be performed. Image processing and analysis address the fundamental issue of allowing an end user to have enhanced information, or statistically more accurate data, of a given image or signal. Thus, mathematical methods are the critical piece in achieving this enhancement.

Given the concepts of time–frequency analysis, the key idea is to decompose the image into its Fourier components. This is easily done with a 2D Fourier transform. Thus, the image becomes a collection of Fourier modes. Figure 14.34 shows a photo and its corresponding 2D Fourier transform. The drawback of the Fourier transform is the fact that the Fourier transform of a sharp-edged object results in a sinc-like function that decays in the Fourier modes like $1/k$, where k is the wavenumber. This slow decay means that localization of the

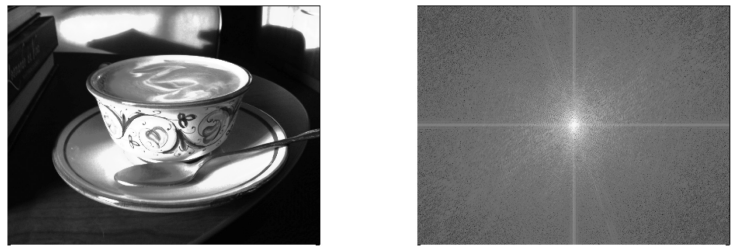


Figure 14.34 Image of a beautifully made cappuccino along with its Fourier transform (log of the absolute value of the spectrum) in two dimensions. The strong vertical and horizontal lines in the spectrum correspond to vertical and horizontal structures in the photo. Looking carefully at the spectrum will also reveal diagonal signatures associated with the diagonal edges in the photo.

image in both the spatial and wavenumber domain is limited. Regardless, the Fourier mode decomposition gives an alternative representation of the image. Moreover, it is clear from the Fourier spectrum that a great number of the Fourier mode components are zero or nearly so. Thus, the concept of image compression can easily be seen directly from the spectrum, i.e. by saving 20% of the dominant modes only, the image can be nearly constructed. This then would compress the image five-fold. As one might also expect, filtering with the Fourier transform can also help process the image to some desired ends.

Wavelets provide a much more sophisticated representation of an image. In particular, wavelets allow for exceptional localization in both the spatial and wavenumber domains. In the wavelet basis, the key quantities are computed from the wavelet coefficients

$$c_\alpha = (u(x,y), \psi_\alpha) \tag{14.8.1}$$

where the wavelet coefficient c_α is equivalent to a Fourier coefficient for a wavelet ψ_α. Since the mid-1980s, the wavelet basis has taken over as the primary tool for image compression due to its excellent space–wavenumber localization properties.

14.8.1 Loading images, additive noise, and Python

To illustrate some of the various aspects of image processing, the following example will load the ideal image, apply Gaussian white noise to each pixel, and show the noisy image. In Python, most standard image formats are easily loaded. The key to performing mathematical manipulations is to then transform the data into double precision numbers that can be transformed, filtered, and manipulated at will. The generic data file uploaded in Python is *uint8*, which is an integer format ranging from 0 to 255. Moreover, colour images have three sets of data for each pixel in order to specify the RGB colour coordinates. The following code uploads an image file (600×800 pixels). Although there are 600×800 pixels, the data are stored as a $600 \times 800 \times 3$ matrix, where the extra dimensions contain the colour coding. This can be made into a 600×800 matrix by turning the image into a black-and-white picture. This is done in the code that follows:

```
from skimage.color import rgb2gray
from skimage import io

A=io.imread('photo.jpeg');
Abw = rgb2gray(A);
```

Note that at the end of the code, the matrices produced are converted to double precision numbers.

To add Gaussian white noise to the images, either the colour or black-and-white versions, the **randn** command is used. In fact, noise is added to both pictures and the output is produced.

```
B=Abw+0.5*np.random.randn(600,800)
```

Note that at the final step, the images are converted back to standard image formats used for JPEG or TIFF images. Figure 14.35 demonstrates the plot produced from the above code. The ability to load and manipulate the data is fundamental to all the image processing applications to be pursued in what follows.

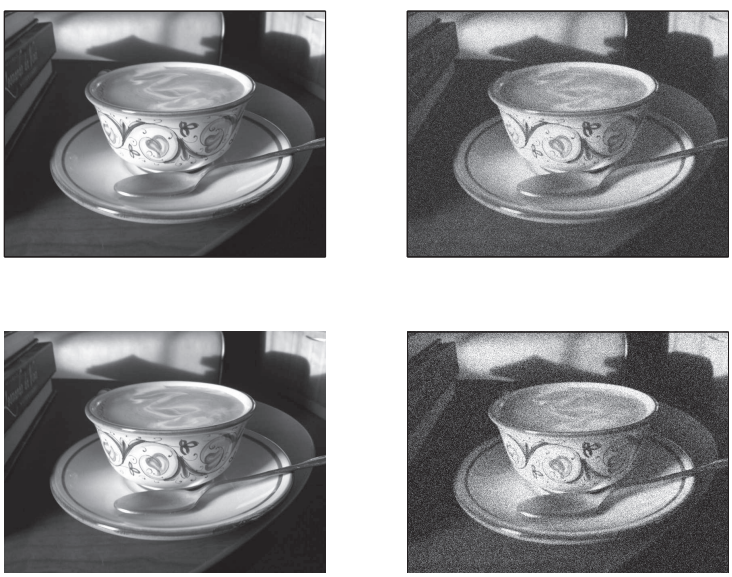


Figure 14.35 Ideal image in colour and black-and-white (left panels) along with their noisy counterparts (right panels).

As a final example of data manipulation, the Fourier transform of the above image can also be considered. Figure 14.34 has already demonstrated the result of this code. But for completeness, it is illustrated here

```
Bt=np.fft.fft2(B)
Bts=fftshift(Bt)
```

Note that in this set of plots, we are once again working with matrices so that commands like **pcolor** are appropriate. These matrices are the subject of image processing routines and algorithms.

As with time–frequency analysis and denoising of time signals, many applications of image processing deal with cleaning up images from imperfections, pixelation, and graininess, i.e. processing of noisy images. The objective in any image processing application is to enhance or improve the quality of a given image. In this section, the filtering of noisy images will be considered with the aim of providing a higher quality, maximally denoised image.

The power of filtering has already been demonstrated in the context of radar detection applications. Ultimately, image denoising is directly related to filtering. To see this, consider Fig. 14.34 of the previous section. In this image, the ideal image is represented along with the log of its Fourier transform. Like many images, the Fourier spectrum is particularly sparse (or nearly zero) for most high-frequency components. Noise, however, tends to generate many high-frequency structures on an image. Thus, it is hoped that filtering of high-frequency components might remove unwanted noise fluctuations or graininess in an image. The top two panels of Fig. 14.36 show an initial noisy image and the log of its Fourier transform. These top two panels can be compared to the ideal image and spectrum shown in Fig. 14.34.

Linear filtering can be applied in the Fourier domain of the image in order to remove the high-frequency scale fluctuations induced by the noise. A simple filter to consider is a Gaussian that takes the form

$$F(k_x, k_y) = \exp(-\sigma_x(k_x - a)^2 - \sigma_y(k_y - b)^2) \tag{14.8.2}$$

where σ_x and σ_y are the filter widths in the x- and y-directions, respectively, and a and b are the centre-frequency values for the corresponding filtering. For the image under consideration, it is a 600×800 pixel image so that the centre-frequency components are located at $k_x = 301$ and $k_y = 401$, respectively.

The **meshgrid** command is used to generate the 2D wavenumbers in the x- and y-directions. This allows for the construction of the filter function that has a 2D form. In particular, a Gaussian centred around $(k_x, k_y) = (301, 401)$ is created with $\sigma_x = \sigma_y = 0.0001$. The bottom two panels in Fig. 14.36 show the filtered spectrum followed by its inverse Fourier transform. The final image on the bottom right of this figure shows the denoised image produced by simple filtering. The filtering significantly improves the image quality.

One can also over-filter and cut out substantial information concerning the figure. Figure 14.37 shows the log of the spectrum of the noisy image for different values of filters in comparison with the unfiltered image. For narrow filters, much of the image information is

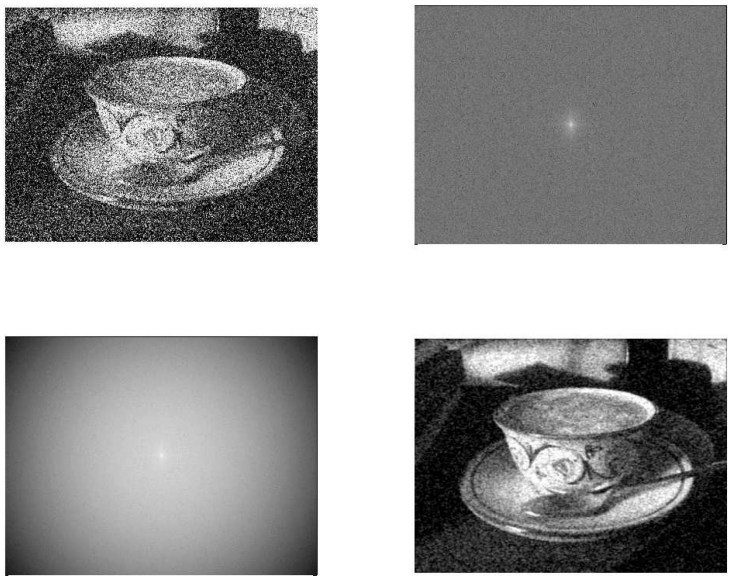


Figure 14.36 A noisy image and its Fourier transform are considered in the top panel. The ideal image is shown in Fig. 14.34. By applying a linear Gaussian filter, the high-frequency noise components can be eliminated (bottom left), and the image quality is significantly improved (bottom right).

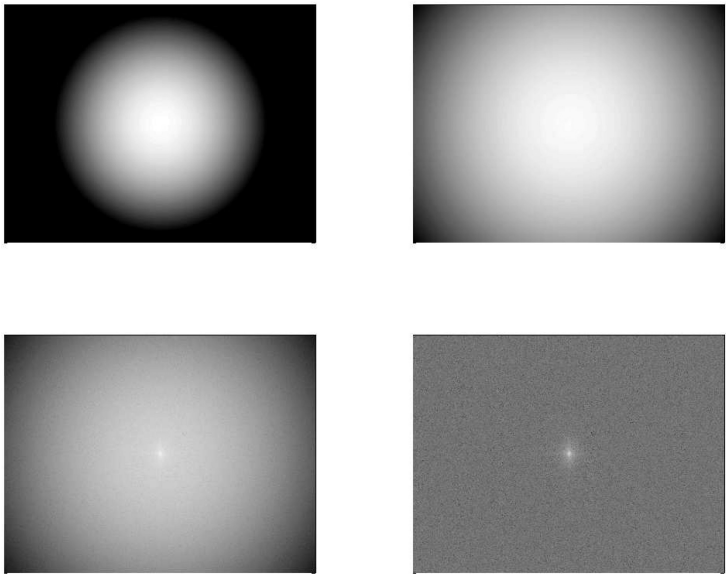


Figure 14.37 Comparison of the log of the Fourier transform for three different values of the filtering strength, $\sigma_x = \sigma_y = 0.01, 0.001, 0.0001$ (top left, top right, bottom left, respectively), and the unfiltered image (bottom right).

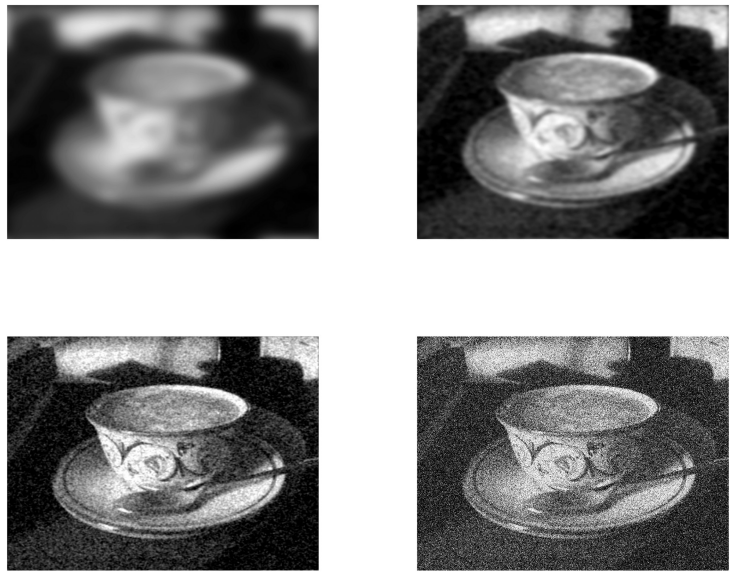


Figure 14.38 Comparison of the image quality for the three filter strengths considered in Fig. 14.37. A high degree of filtering blurs the image and no fine scale features are observed (top left). In contrast, moderate strength filtering produces exceptional improvement of the image quality (top right and bottom left). These denoised images should be compared to the unfiltered image (bottom right).

irretrievably lost along with the noise. Finding an optimal filter is part of the image processing agenda. In the image spatial domain, the filtering strength (or width) can be considered. Figure 14.38 shows a series of filtered images and their comparison to the unfiltered image. Strong filtering produces a smooth, yet blurry image. In this case, all fine-scale features are lost. Moderate levels of filtering produce reasonable images with significant reduction of the noise in comparison to the nonfiltered image.

Gaussian filtering is only one type of filtering that can be applied. There are, just like in signal processing applications, myriads of filters that can be used to process a given image, including low-pass, high-pass, band-pass, etc. As a second example filter, the Shannon filter is considered. The Shannon filter is simply a step function with value of unity within the transmitted band and zero outside of it. In the example that follows, a Shannon filter is applied of width 50 pixels around the centre frequency of the image.

```
Fs=np.zeros((600,800))
width=50
mask=np.ones((2*width+1,2*width+1))
Fs[301-width:301+width+1,401-width:401+width+1]=mask
Btsf=np.multiply(Bts,Fs)
Btf=fftshift(Btsf)
Bf=np.fft.ifft2(Btf)
```

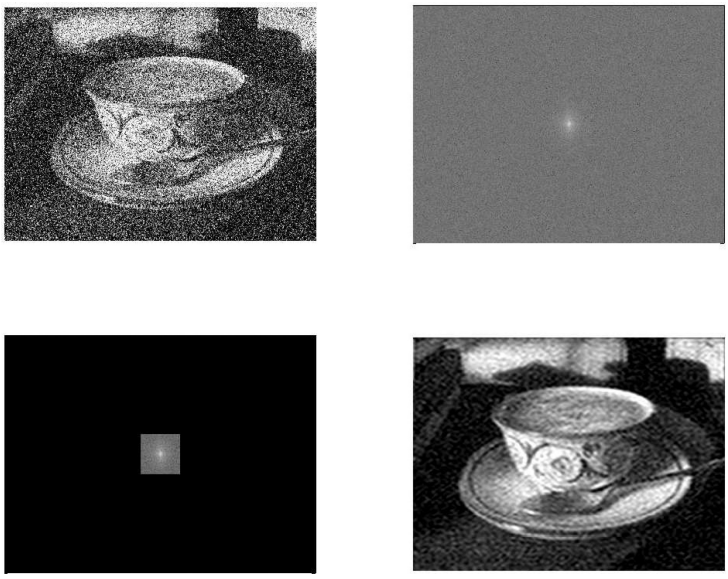


Figure 14.39 A noisy image and its Fourier transform are considered in the top panel. The ideal image is shown in Fig. 14.34. By applying a Shannon (step function) filter, the high-frequency noise components can be eliminated (bottom left) and the image quality is significantly improved (bottom right).

Figure 14.39 is a companion to Fig. 14.36. The only difference between them is the filter chosen for the image processing. The key difference is represented in the lower left panel of both figures. Note that the Shannon filter simply suppresses all frequencies outside of a given filter box. The performance difference between Gaussian filtering and Shannon filtering appears to be fairly marginal. However, there may be some applications where one or the other is more suitable. The image enhancement can also be explored as a function of the Shannon filter width. Figure 14.40 shows the image quality as the filter is widened along with the unfiltered image. Strong filtering again produces a blurred image, while moderate filtering produces a greatly enhanced image quality.

The width is adjusted by considering the number of filter pixels around the centre frequency with value unity. One might be able to argue that the Gaussian filter produces slightly better image results since the inverse Fourier transform of a step function filter produces sinc-like behaviour.

14.9 Diffusion and Image Processing

Filtering is not the only way to denoise an image. Intimately related to filtering is the use of diffusion for image enhancement. Consider for the moment the simplest spatial diffusion process in two dimensions, i.e. the heat equation:

$$u_t = D\nabla^2 u \tag{14.7.1}$$

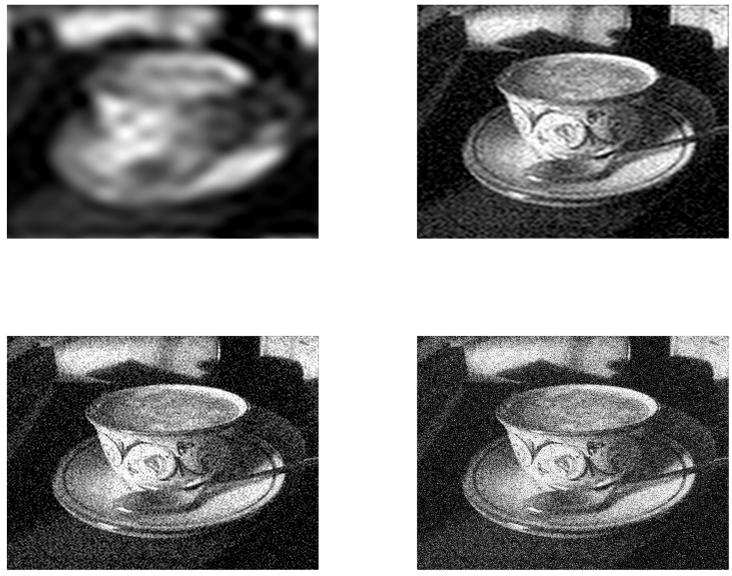


Figure 14.40 Comparison of the image quality for the three filter widths. A high degree of filtering blurs the image and no fine scale features are observed (top left). In contrast, moderate strength filtering produces exceptional improvement of the image quality (top right and bottom left). These denoised images should be compared to the unfiltered image (bottom right).

where $u(x, y)$ will represent a given image, $\nabla^2 = \partial_x^2 + \partial_y^2$, D is a diffusion coefficient, and some boundary conditions must be imposed. If for the moment, we consider periodic boundary conditions, then the solution to the heat equation can be found from, for instance, the Fourier transform

$$\hat{u}_t = -D(k_x^2 + k_y^2)\hat{u} \quad \rightarrow \quad \hat{u} = \hat{u}_0 e^{-D(k_x^2+k_y^2)t} \tag{14.7.2}$$

where $\hat{u}$ is the Fourier transform of $u(x, y)$ and $\hat{u}_0$ is the Fourier transform of the initial conditions, i.e. the original noisy image. The solution of the heat equation illustrates a key and critical concept: the wavenumbers (spatial frequencies) decay according to a Gaussian function. Thus, linear filtering with a Gaussian is equivalent to a linear diffusion of the image for periodic boundary conditions.

The above argument establishes some equivalency between filtering and diffusion. However, the diffusion formalism provides a more general framework in which to consider image clean-up since the heat equation can be modified to

$$u_t = \nabla \cdot (D(x, y)\nabla u) \tag{14.7.3}$$

where $D(x, y)$ is now a spatial diffusion coefficient. In particular, the diffusion coefficient could be used to great advantage to target trouble spots on an image while leaving relatively noise-free patches alone.

To solve the heat equation numerically, we discretize the spatial derivative with a second-order scheme (see Table 3.1) so that

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{\Delta x^2}\left[u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)\right] \tag{14.7.4a}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{1}{\Delta y^2}\left[u(x, y+\Delta y) - 2u(x,y) + u(x, y-\Delta y)\right]\,. \tag{14.7.4b}$$

This approximation reduces the partial differential equation to a system of ordinary differential equations. Once this is accomplished, then a variety of standard time-stepping schemes for differential equations can be applied to the resulting system.

14.9.1 The ODE system for 1D diffusion

Consider first diffusion in one dimension so that $u(x, y) = u(x)$. The vector system for Python can then be formulated. To define the system of ODEs, we discretize and define the values of the vector $\mathbf{u}$ in the following way:

$$\begin{aligned}
&u(-L) = u_1\\
&u(-L+\Delta x) = u_2\\
&\vdots\\
&u(L-2\Delta x) = u_{n-1}\\
&u(L-\Delta x) = u_n\\
&u(L) = u_{n+1}\,.
\end{aligned}$$

If periodic boundary conditions, for instance, are considered, then periodicity requires that $u_1 = u_{n+1}$. Thus, the system of differential equations solves for the vector

$$\mathbf{u} = \begin{pmatrix} u_1\\ u_2\\ \vdots\\ u_n \end{pmatrix}\,. \tag{14.7.5}$$

The governing heat equation is then reformulated in discretized form as the differential equation system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\Delta x^2}\mathbf{A}\mathbf{u}\,, \tag{14.7.6}$$

where $\mathbf{A}$ is given by the sparse matrix

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & & & \vdots \\ & & & & & 0 \\ \vdots & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}, \tag{14.7.7}$$

and the values of one on the upper right and lower left of the matrix result from the periodic boundary conditions. The system of differential equations can now be easily solved with a standard time-stepping algorithm.

The basic algorithm would be as follows:

(1) Build the sparse matrix $\mathbf{A}$.

```
e1 = np.ones((n, 1))
diagonals = [e1.flatten(), -2 * e1.flatten(), e1.flatten()]
offsets = [-1, 0, 1]
A = spdiags(diagonals, offsets, n, n, format='csr')
A[0, -1] = 1
A[-1, 0] = 1
```

(2) Generate the desired initial condition vector $\mathbf{u} = \mathbf{u}_0$.

(3) Call an ODE solver from the Python suite. The matrix $\mathbf{A}$, the diffusion constant κ, and spatial step Δx need to be passed into this routine.

```
y = odeint(rhs, u0, tspan, args=(k,dx,A))
```

The function *rhs.m* should be of the following form:

```
def shoot2(u, dummy, k, dx, A):
    return (k/dx/dx)*A*u
```

(4) Plot the results as a function of time and space.

The algorithm is thus fairly routine and requires very little effort in programming since we can make use of the standard time-stepping algorithms already available in Python.

14.9.2 2D Python implementation

In the case of two dimensions, the calculation becomes slightly more difficult since the 2D data are represented by a matrix and the ODE solvers require a vector input for the initial data. For this case, the governing equation is

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right). \tag{14.7.8}$$

Provided $\Delta x = \Delta y = \delta$ are the same, the system can be reduced to the linear system

$$\frac{d\mathbf{u}}{dt} = \frac{\kappa}{\delta^2} \mathbf{A}\mathbf{u}\,, \tag{14.7.9}$$

where we have arranged the vector $\mathbf{u}$ by stacking slices of the data in the second dimension y. This stacking procedure is required for implementation purposes of Python. Thus, by defining

$$u_{nm} = u(x_n, y_m) \tag{14.7.10}$$

the collection of image (pixel) points can be arranged as follows:

$$\mathbf{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}. \tag{14.7.11}$$

The matrix $\mathbf{A}$ is a sparse matrix, and so the sparse implementation of this matrix can be used advantageously in Python.

Again, the system of differential equations can now be easily solved with a standard time-stepping algorithm such as *ode23* or *ode45*. The basic algorithm follows the same course as the 1D case, but extra care is taken in arranging the 2D data into a vector.

14.9.3 Diffusion of an image

To provide a basic implementation of the above methods, consider the following Python code which loads a given image file, converts it to double precision numbers, then diffuses the image in order to remove some of the noise content. The process begins once again with the loading of an ideal image on which noise will be projected. As before, the black-and-white version of the image will be the object consideration for diagnostic purposes. Figure 14.41



Figure 14.41 A beautiful looking image demonstrating the classic fernleaf pattern in a cappuccino (left panel). By adding noise to the ideal image, we are forced to use mathematics to clean up this image (right panel).

shows the ideal image along with a noisy black-and-white version. Diffusion will be used to denoise this image.

In what follows, a constant diffusion coefficient will be considered. To make the Laplacian operator in two dimensions, the **kron** command is used. The generated operator **L** takes two derivatives in x and y and is the Laplacian in two dimensions. This operator is constructed to act upon data that have been stacked as in (14.7.11).

It simply remains to evolve the image through the diffusion equation. But now that the Laplacian operator has been constructed, it simply remains to reshape the image, determine a time-span for evolving, and choose a diffusion coefficient. The diffusion of the image produces a plot that tracks the image from its initial state to the final diffused image.

The above code calls on the function **image_rhs**, which contains the diffusion equation information. Figure 14.42 shows the evolution of the image through the diffusion process. Note that only a slight amount of diffusion is needed, i.e. a small diffusion constant, as well as a short diffusion time, before the pixelation has been substantially reduced. Continued diffusion starts to degrade image quality. This corresponds to over-filtering the image.

14.9.4 Nonlinear filtering and diffusion

As mentioned previously, the linear diffusion process is equivalent to the application of a Gaussian filter. Thus, it is not clear that the diffusion process is any better than simple filtering. However, the diffusion process has several distinct advantages: first, particular regions in the spatial figure can be targeted by modification of the diffusion coefficient $D(x, y)$. Second, nonlinear diffusion can be considered for enhancing certain features of the image. This corresponds to a nonlinear filtering process. In this case,

$$u_t = \nabla \cdot (D(u, \nabla u)\nabla u) \tag{14.7.12}$$

where the diffusion coefficient now depends on the image and its gradient. Nonlinear diffusion, with a properly constructed D above, can be used, for instance, as an effective method for extracting edges from an image [53]. Thus, although this section has only considered a simple linear diffusion model, the ideas are easily generalized to account for more general concepts and image-processing aims.

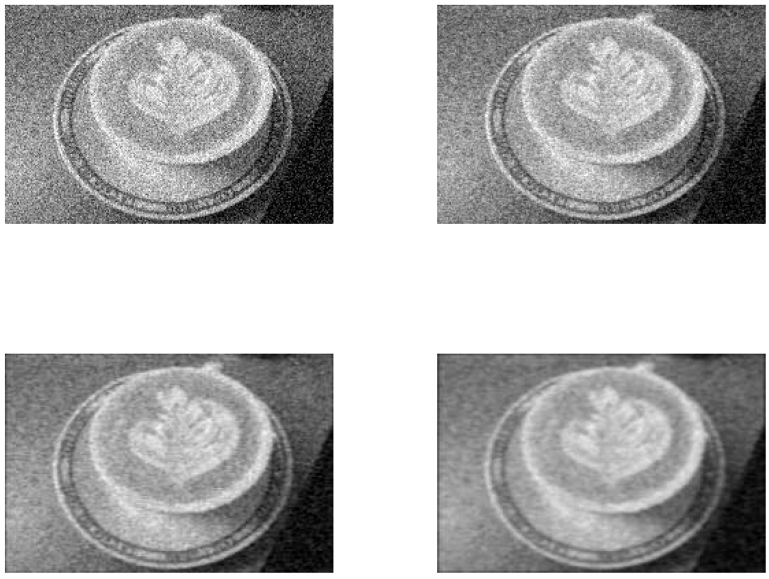


Figure 14.42 Image denoising process for a diffusion constant of $D = 0.0005$ as a function of time for $t = 0, 0.005, 0.02$, and 0.04. Perhaps the best image is for $t = 0.02$, where a moderate amount of diffusion has been applied. Further diffusion starts to degrade the image quality. The image was rescaled to 160×240 pixels.

14.10 Compressive Sensing and Circumventing Nyquist

Compressed sensing has a significant connection with spectral and/or wavelet representations. Specifically, we have shown that many images are naturally sparse in the Fourier or wavelet domain, which allows for compression algorithms. Compressive sensing is a signal recovery problem. The idea is to exploit the sparse nature of images in order to reconstruct them using greatly reduced sampling. These concepts also make natural connection to image compression algorithms. What is of particular importance is choosing a basis representation in which images are, in fact, sparse. Thus, the connection to Fourier and wavelet theory.

It is informative to consider the reconstruction of a temporal signal, and the use of the L^1 norm, which is a proxy for sparsity. This example [54] makes connection directly to signal processing. Consider the "A" key on a touch-tone telephone, which is the sum of two sinusoids with incommensurate frequencies [54]:

$$f(t) = \sin(1394\pi t) + \sin(3266\pi t). \tag{14.10.1}$$

To begin, the signal will be sampled over $1/8$ of a second at $40{,}000\,\mathrm{Hz}$. Thus, a vector of length 5000 will be generated. The signal can also be represented using the discrete cosine transform (DCT). In Python, the following lines are sufficient to produce the signal and its DCT:

```
from scipy.fftpack import dct
n = 5000
t = np.linspace(0, 1/8, n)
f = np.sin(1394*np.pi*t) + np.sin(3266*np.pi*t)
ft = dct(f)
```

As can be seen from the form of the signal, there should be dominance by two Fourier modes in (14.7.13). However, since the frequencies are incommensurate and do not fall within the frequencies spanned by the DCT, there are a number of nonzero DCT coefficients. Figures 14.43 and 14.44 show the original signal over $1/8$ of a second and a blow-up of the signal in both time and frequency. The key observation to make: *The signal is highly sparse in the frequency domain*.

The objective now will be to randomly *sample* the signal at a much lower frequency (i.e. sparse sampling) and try to reconstruct the original signal, in both frequency and time, as best as possible. Such an exercise is at the heart of compressive sensing. Namely, given that the signal is sparse (in frequency) to begin with, can we sample sparsely and yet faithfully reconstruct the desired signal [55–57]? Indeed, compressive sensing algorithms are data acquisition protocols that perform as if, where possible, to directly acquire just the important

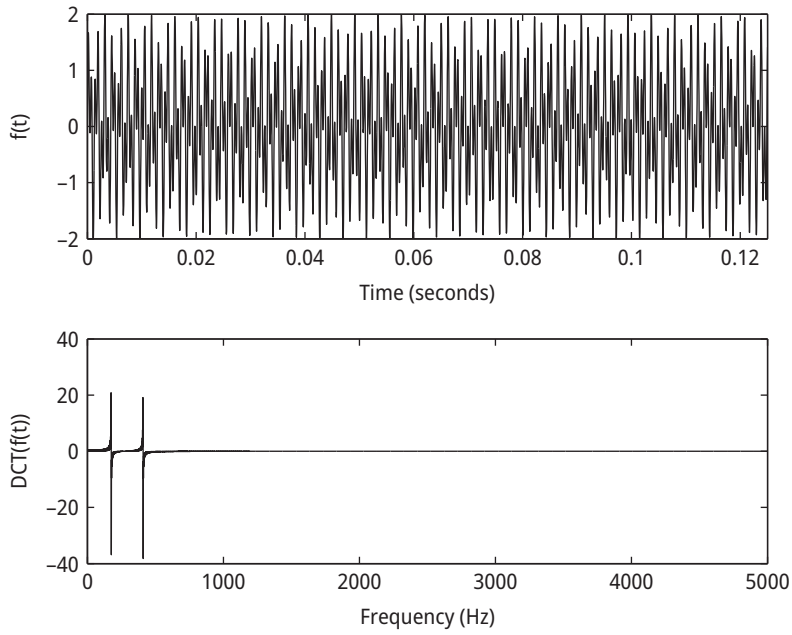


Figure 14.43 Original "A"-tone signal and its discrete cosine transform sampled at 40,000 Hz for 1/8 seconds.

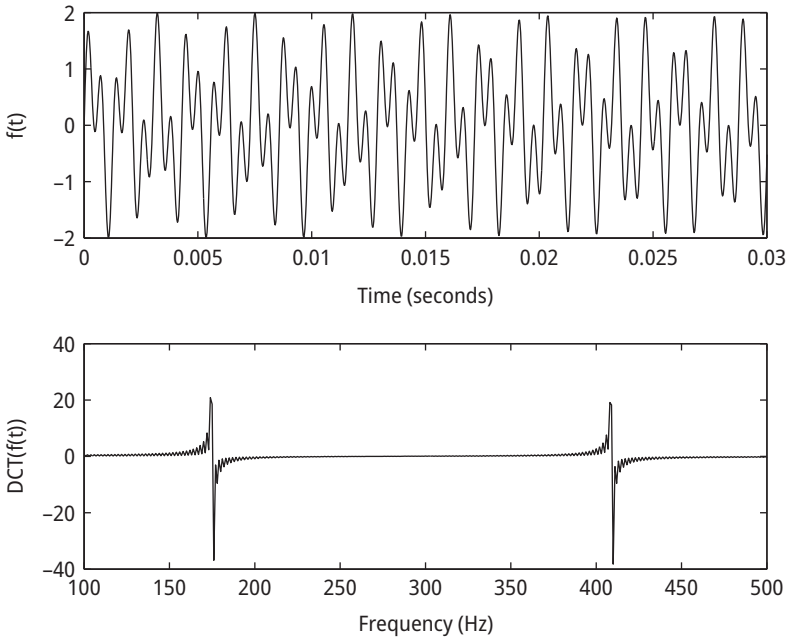


Figure 14.44 Zoom-in of the original "A"-tone signal and its discrete cosine transform sampled at 40,000 Hz for 1/8 seconds.

information about the signal, i.e. the sparse signal representation in the DCT domain. In effect, the idea is to acquire only the important information and ignore that part of the data, which effectively makes no contribution to the signal. Figure 14.43 clearly shows that most of the information in the original signal can be neglected when considered from the perspective of the DCT domain. Thus, the sampling algorithm must find a way to key in on the prominent features of the signal in the frequency domain. Since the L^1 norm promotes sparsity, this will be the natural basis in which to work to reconstruct the signal from our sparse sampling [55–57].

14.10.1 The matrix problem

Before proceeding too far along, it is important to relate the current exercise to the linear algebra problems considered in the previous section. Specifically, this signal reconstruction problem is nothing more than a large underdetermined system of equations. To be more precise, the conversion from the time domain to frequency domain via the DCT can be thought of as a linear transformation

$$\psi \mathbf{c} = \mathbf{f} \tag{14.10.2}$$

where $\mathbf{f}$ is the signal vector in the time domain (plotted in the top panels of Figs. 14.43 and 14.44) and $\mathbf{c}$ are the cosine transform coefficients representing the signal in the DCT domain

(plotted in the bottom panels of Figs. 14.43 and 14.44). The matrix ψ represents the DCT transform itself. The key observation is that most of the coefficients of the vector $\mathbf{c}$ are zero, i.e. it is sparse as clearly demonstrated by the dominance of the two cosine coefficients at $\approx$175 Hz and $\approx$410 Hz. Note that the matrix ψ is of size $n \times n$, while $\mathbf{f}$ and $\mathbf{c}$ are $n \times 1$ vectors. For the example plotted, $n = 5000$.

The choice of basis functions is critical in carrying out the compressed sensing protocol. In particular, the signal must be sparse in the chosen basis. For the example here of a cosine basis, the signal is clearly sparse, allowing us to accurately reconstruct the signal using sparse sampling. The idea is to now sample the signal randomly (and sparsely) so that

$$\mathbf{b} = \phi\mathbf{f} \tag{14.10.3}$$

where $\mathbf{b}$ is a few (m) random samples of the original signal $\mathbf{f}$ (ideally $m \ll n$). Thus, ϕ is a subset of randomly permuted rows of the identity operator. More complicated sampling can be performed, but this is a simple example that will illustrate all the key features. Note here that $\mathbf{b}$ is an $m \times 1$, vector while the matrix ϕ is of size $m \times n$.

Approximate signal reconstruction can then be performed by solving the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{14.10.4}$$

where $\mathbf{b}$ is an $m \times 1$ vector, $\mathbf{x}$ is $n \times 1$ vector and

$$\mathbf{A} = \phi\psi \tag{14.10.5}$$

is a matrix of size $m \times n$. Here, $\mathbf{x}$ is the sparse approximation to the full DCT coefficient vector. Thus, for $m \ll n$, the resulting linear algebra problem is highly underdetermined as in the previous section. The idea is then to solve the underdetermined system using an appropriate norm constraint that best reconstructs the original signal. As already demonstrated, the L^1 norm promotes sparsity and is highly appropriate given the sparsity already demonstrated. The signal reconstruction is performed by using

$$\mathbf{f} \approx \psi\mathbf{x}. \tag{14.10.6}$$

If the original signal had exactly m nonzero coefficients, the reconstruction could be made exact.

14.10.2 Signal reconstruction

To begin the reconstruction process, the signal must first be randomly sampled. In this example, the original signal $\mathbf{f}$ with $n = 5000$ points will be sampled at 10% so that the vector $\mathbf{b}$ above will have $m = 500$ points. The following code randomly rearranges the integers 1 to 5000 (**randintrlv**) so that the vector **perm** retrieves 500 random sample points.

```
m = 500
r1 = np.random.permutation(n)
perm = r1[:m]

f2 = f[perm]
t2 = t[perm]
```

In this example, the resulting vectors $\mathbf{t2}$ and $\mathbf{f2}$ are the 500 random point locations (out of the 5000 original). The $m \times n$ matrix $\mathbf{A}$ is then constructed by constructing $\mathbf{A} = \phi\psi$.

```
D = dct(np.eye(n))
A = D[perm, :]
```

Recall that the matrix ψ was the DCT transform, while the matrix ϕ was the permutation matrix of the identity yielding the sampling points for $\mathbf{f2}$.

Although the L^1 norm is of primary importance, the underdetermined system will be solved in three ways: with the pseudo-inverse, the backslash, and with L^1 optimization. It was already illustrated in the previous section that the backslash promoted sparsity for such systems, but as will be illustrated, all sparsity is not the same! The following commands generate all three solutions for the underdetermined system.

```
x = np.linalg.pinv(A) @ f2
x2 = np.linalg.lstsq(A, f2, rcond=None)[0]
x3 = cp.Variable(n)
objective = cp.Minimize(cp.norm(x3, 1))
constraints = [A @ x3 == f2]
problem = cp.Problem(objective, constraints)
problem.solve()
```

The above code generates three vectors approximating the DCT coefficients, i.e. these solutions are used for the reconstruction (14.7.18). Figure 14.45 illustrates the results of the reconstructed signal in the DCT domain over the range of 100–500 Hz. The top panel illustrates the original signal in the DCT domain, while the next three panels represent the approximation to the DCT coefficients using the L^1 norm optimization, the pseudo-inverse, and the backslash, respectively. As expected, both the L^1 and backslash methods produce sparse representations. However, the backslash produces a result that looks nothing like the original DCT signal. In contrast, the pseudo-inverse does not produce a sparse result.

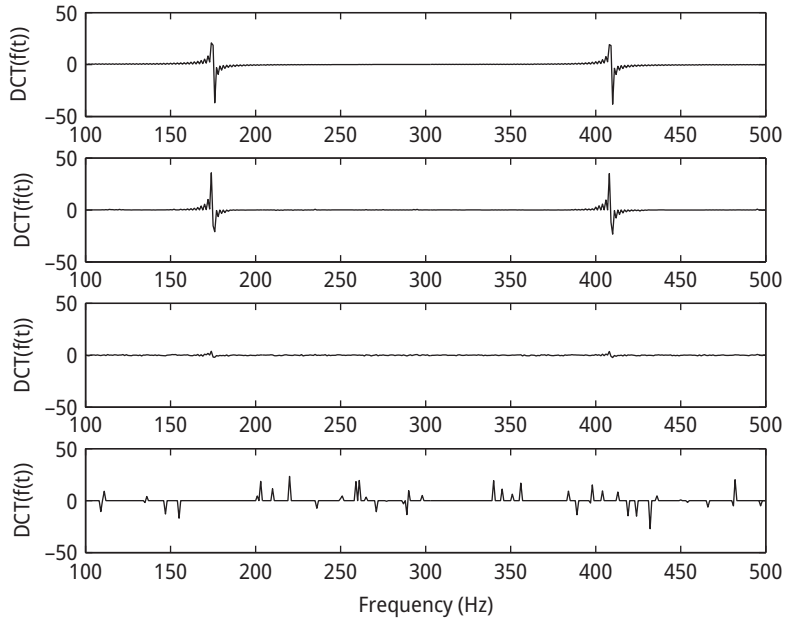


Figure 14.45 Original signal in the DCT domain (top panel) and the reconstructions using L^1 optimization (second panel), the pseudo-inverse (third panel), and the backslash (fourth panel), respectively. Note that the L^1 optimization does an exceptional job at reconstructing the spectral content with only 10% sampling. In contrast, both the pseudo-inverse and backslash fail miserably at the task.

Indeed, the spectral content is quite full, and only the slightest indication of the importance of the cosine coefficients at ≈175 Hz and ≈410 Hz is given.

To more clearly see the results of the cosine coefficient reconstruction, Fig. 14.46 zooms in near the ≈410 Hz peak of the spectrum. The top panel shows the excellent reconstruction yielded by the L^1 optimization routine. The pseudo-inverse (middle panel) and backslash fail to capture any of the key features of the spectrum. This example shows the almost "magical" ability of the L^1 optimization to reconstruct the original signal in the cosine domain.

It remains to consider the signal reconstruction in the time domain. To perform this task, the DCT domain data must be transformed to the time domain via the discrete cosine transform:

```
sig1 = dct(x)
sig2 = dct(x2)
sig3 = dct(x3.value)
```

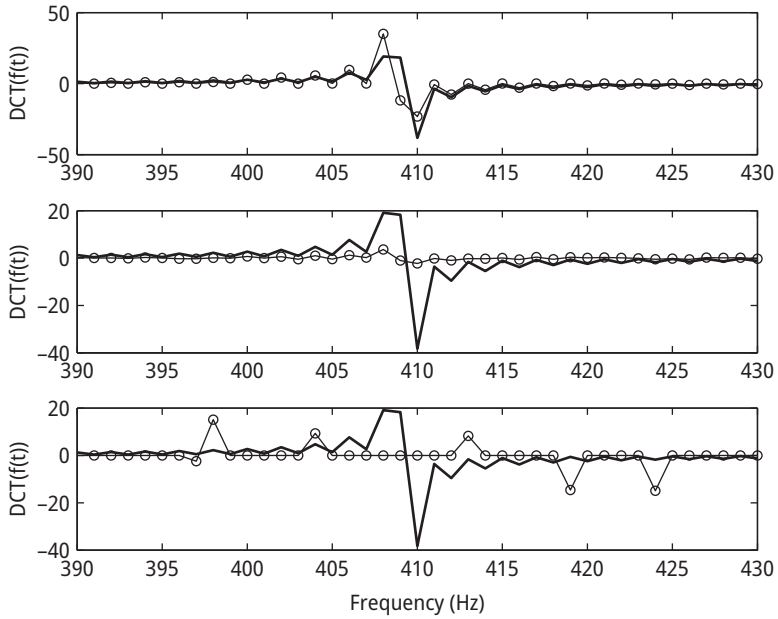


Figure 14.46 Zoom-in of the DCT domain reconstructions using L^1 optimization (top panel), the pseudo-inverse (middle panel), and the backslash (bottom panel), respectively. Here, a direct comparison is given to the original signal (bold line). Although both the L^1 optimization and backslash produce sparse results, it is clear the backslash produces a horrific reconstruction of the signal.

As before, the L^1 optimization will be compared to the standard L^2 schemes of the pseudo-inverse and backslash. Figure 14.47 illustrates the original signal (top panel) along with the L^1 optimization, pseudo-inverse and backslash methods, respectively. Even at 10% sampling, the L^1 does a very nice job in reconstructing the signal. In contrast, both L^2-based methods fail severely in the reconstruction. Note, however, that in this case, the pseudo-inverse does promote sparsity in the time domain. Indeed, when the original sparse sampling points are overlaid on the reconstruction, it is clear that the reconstruction does a very nice job of minimizing the L^2 error for these points. In contrast, the backslash produces a solution that almost resembles white noise. However, as with the pseudo-inverse, if considering only the sampling points, the L^2 error is quite low.

Finally, Fig. 14.48 demonstrates the effect of sampling on the signal reconstruction. In what was considered in the previous plots, the sampling was $m = 500$ points out of a possible $n = 5000$, thus 10% sparse sampling was sufficient to approximate the signal. The sampling is lowered to see the deterioration of the signal reconstructions. In particular, Fig. 14.48 demonstrates the signal reconstruction using $m = 100$ (second panel),

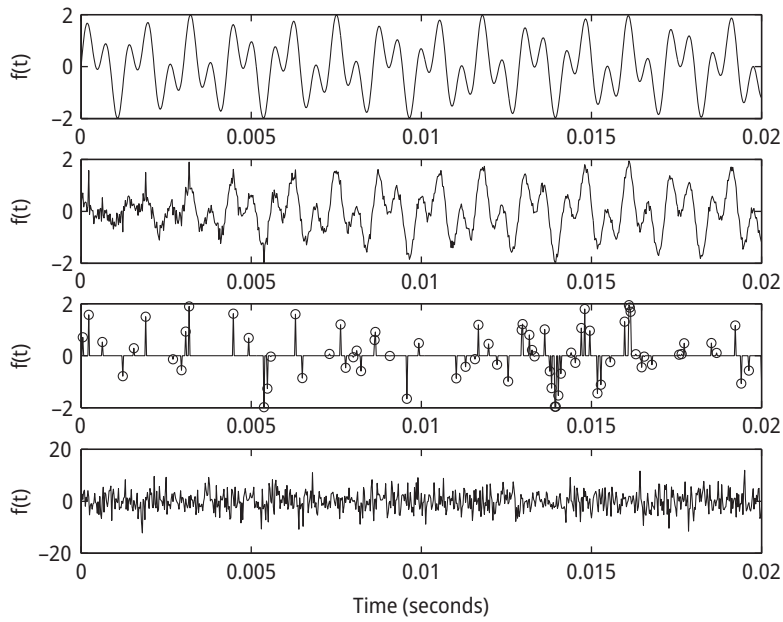


Figure 14.47 Original time-domain signal (top panel) and the reconstructions using L^1 optimization (second panel), the pseudo-inverse (third panel), and the backslash (fourth panel), respectively. Note that the L^1 optimization does an exceptional job at reconstructing the signal with only 10% sampling. In contrast, both the pseudo-inverse and backslash fail. Note that in the pseudo-inverse reconstruction (third panel), the original sparse sampling points are included, demonstrating that the method does well in keeping the L^2 error in check for these points. However, the L^2 methods do poorly elsewhere.

300 (third panel), and 500 (bottom panel) sampling points. In the top panel, the original signal is included for reference. In subsequent panels, the original signal is included as the dotted line. Also included is the sparse sample points (circles). This provides a naked-eye measure of the effectiveness of the compressed sensing algorithm and L^1 optimization method. At $m = 500$ (bottom panel), the reconstruction is quite faithful to the original signal. As m is lowered, the signal recovery becomes worse. But remarkably at even 2% sampling $(m = 100)$, many of the key features are still visible in the signal reconstruction. This is especially remarkable given the amazing sparsity in the sampling. For instance, between ≈ 0.006 and 0.012 Hz, there are no sampling points, yet the compressed sensing scheme still picks up the key oscillatory features in this range. This is in direct contrast to the thinking and intuition established by Nyquist. Indeed, Nyquist (or Nyquist–Shannon sampling theory) states that to completely recover a signal, one must sample at twice the rate of the highest frequency of the signal. This is the gold standard of the information theory community. Here, however, the compressed sensing algorithm seems to be Nyquist. The apparent contradiction is overcome by one simple idea: sparsity. Specifically, Nyquist–Shannon theory assumes in its formulation that the signal is dense in the frequency domain. Thus, it is certainly true

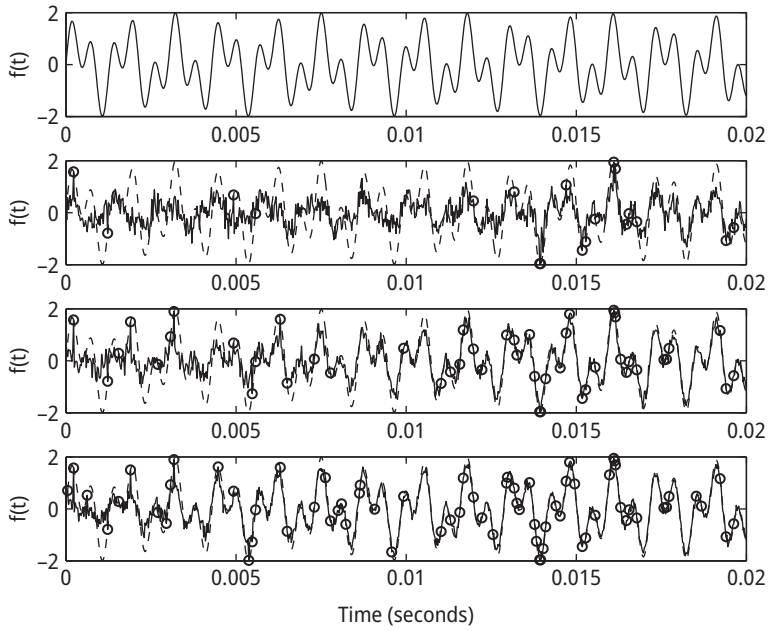


Figure 14.48 The original signal (top panel) and the reconstruction using $m = 100$ (second panel), 300 (third panel), and 500 (bottom panel) sampling points. In the top panel, the original signal is included for reference. In subsequent panels, the original signal is included as the dotted line. Also included is the sparse sample points (circles).

that an accurate reconstruction can only be obtained by sampling at twice the highest frequency. Here, however, compressed sensing worked on a fundamental assumption about the sparsity of the signal. This is a truly amazing idea that is having revolutionary impact across many disciplines of science [55–57].

15 Matrix Decompositions

15.1 The Singular Value Decomposition (SVD) 403
15.1.1 Computing the SVD 406
15.2 The SVD in Broader Context 408
15.2.1 Eigenvalues, eigenvectors, and diagonalization 408
15.2.1.1 Option I 408
15.2.1.2 Option II 409
15.2.2 Diagonalization via SVD 411
15.2.3 Useful theorems of SVD 411
15.2.4 Low-dimensional reductions 413
15.3 Introduction to Principal Component Analysis (PCA) 414
15.3.1 Data collection and ordering 416
15.3.2 The covariance matrix 417
15.3.2.1 Achieving the goal 418
15.4 Principal Components, Diagonalization, and SVD 418
15.4.1 Diagonalization 419
15.4.1.1 Eigenvectors and eigenvalues 419
15.4.1.2 Singular value decomposition 420
15.4.2 Spring experiment 421
15.5 Principal Components and Proper Orthogonal Modes 422
15.5.1 Summary and limits of SVD/PCA/POD 428
15.6 Robust PCA 429
15.6.1 Matrix decomposition: Low-rank plus sparse 430
15.7 Dynamic Mode Decomposition (DMD) 436
15.7.1 pyDMD package 439
15.8 Koopman Operators 442
15.9 Randomized Linear Algebra and Scalable SVD and DMD 448
15.10 Autoencoders and Nonlinear SVD 451
15.11 Shallow Recurrent Decoder (SHRED) and Nonlinear SVD 458

Linear algebra plays a central role in almost every application area of mathematics in the physical, engineering, and biological sciences. It is perhaps the most important theoretical framework to be familiar with as a student of mathematics. Thus, it is no surprise that it also plays a key role in data analysis and computation. In this chapter, emphasis will be placed

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0016

squarely on matrix decomposition methods, and in particular the *singular value decomposition* (SVD). This concept is often untouched in undergraduate courses in linear algebra, yet it forms one of the most powerful techniques for analysing a myriad of application areas. Moreover, there are variants of SVD, both linear and nonlinear, that allow for even more flexible characterizations of data.

15.1 The Singular Value Decomposition (SVD)

In even the earliest experience of linear algebra, the concept of a matrix transforming a vector via multiplication was defined. For instance, the vector $\mathbf{x}$ when multiplied by a matrix $\mathbf{A}$ produces a new vector $\mathbf{y}$ that is now aligned, generically, in a new direction with a new length. To be more specific, the following example illustrates a particular transformation:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \quad \rightarrow \quad \mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}. \tag{15.1.1}$$

Figure 15.1(a) shows the vector $\mathbf{x}$ and its transformed version, $\mathbf{y}$, after application of the matrix $\mathbf{A}$. Thus, generically, matrix multiplication will rotate and stretch (compress) a given vector as prescribed by the matrix $\mathbf{A}$ (see Fig. 15.1(a)).

The rotation and stretching of a transformation can be precisely controlled by proper construction of the matrix $\mathbf{A}$. In particular, it is well known that in a two-dimensional space, the rotation matrix

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{15.1.2}$$

takes a given vector $\mathbf{x}$ and rotates it by an angle θ to produce the vector $\mathbf{y}$. The transformation produced by $\mathbf{A}$ is known as a *unitary transformation* since the matrix inverse is $\mathbf{A}^{-1} = \bar{\mathbf{A}}^T$, where the bar denotes complex conjugation [58]. Thus, rotation can be directly specified without the vector being scaled. To scale the vector in length, the matrix

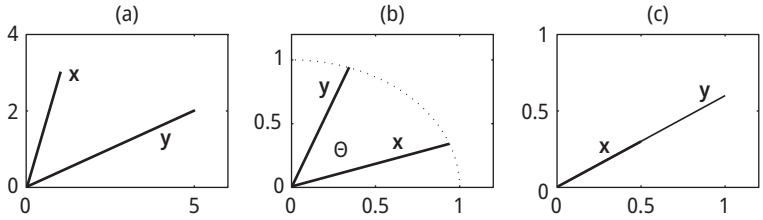


Figure 15.1 Transformation of a vector $\mathbf{x}$ under the action of multiplication by the matrix $\mathbf{A}$, i.e. $\mathbf{y} = \mathbf{A}\mathbf{x}$. (a) Generic rotation and stretching of the vector as given by (15.1.1). (b) Rotation by 50° of a unit vector by the rotation matrix (15.1.2). (c) Stretching of a vector to double its length using (15.1.3) with $\alpha = 2$.

$$\mathbf{A} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \tag{15.1.3}$$

can be applied to the vector $\mathbf{x}$. This multiplies the length of the vector $\mathbf{x}$ by α. If $\alpha = 2$ (0.5), then the vector is twice (half) its original length. The combination of the above two matrices gives arbitrary control of rotation and scaling in a two-dimensional vector space. Figure 15.1 demonstrates some of the various operations associated with the above matrix transformations.

A *singular value decomposition* (SVD) is a factorization of a matrix into a number of constitutive components, all of which have a specific meaning in applications. The SVD, much as illustrated in the preceding paragraph, is essentially a transformation that stretches/compresses and rotates a given set of vectors. In particular, the following geometric principle will guide our forthcoming discussion: the image of a unit sphere under any $m \times n$ matrix is a hyper-ellipse. A hyper-ellipse in $\mathbb{R}^m$ is defined by the surface obtained upon stretching a unit sphere in $\mathbb{R}^m$ by some factors $\sigma_1, \sigma_2, \cdots, \sigma_m$ in the orthogonal directions $\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_m \in \mathbb{R}^m$. The stretchings σ_i can possibly be zero. For convenience, consider the $\mathbf{u}_j$ to be unit vectors so that $\|\mathbf{u}_j\|_2 = 1$. The quantities $\sigma_j \mathbf{u}_j$ are then the *principal semi-axes* of the hyper-ellipse with the length σ_j. Figure 15.2 demonstrates a particular hyper-ellipse created under the matrix transformation $\mathbf{A}$ in $\mathbb{R}^2$.

A few things are worth noting at this point. First, if $\mathbf{A}$ has rank r, exactly r of the lengths σ_j will be nonzero. And if the matrix $\mathbf{A}$ is an $m \times n$ matrix, where $m > n$, at most n of the σ_j will be nonzero. Consider for the moment a full rank matrix $\mathbf{A}$. Then the n singular values of $\mathbf{A}$ are the lengths of the principal semi-axes $\mathbf{A}S$ as shown in Fig. 15.2. Convention assumes that the singular values are ordered with the largest first and then in descending order: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n > 0$.

On a more formal level, the transformation from the unit sphere to the hyper-ellipse can be more succinctly stated as follows:

$$\mathbf{A}\mathbf{v}_j = \sigma_j \mathbf{u}_j \quad 1 \leq j \leq n\,. \tag{15.1.4}$$

Thus, in total, there are n vectors that are transformed under $\mathbf{A}$. A more compact way to write all of these equations simultaneously is with the representation

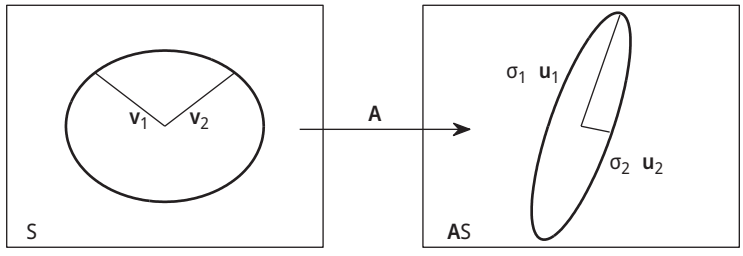


Figure 15.2 Image of a unit sphere S transformed into a hyper-ellipse $\mathbf{A}S$ in $\mathbb{R}^2$. The values of σ_1 and σ_2 are the singular values of the matrix $\mathbf{A}$ and represent the lengths of the semi-axes of the ellipse.

$$\begin{bmatrix} & & \\ & \mathbf{A} & \\ & & \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} & & & \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \\ & & & \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \tag{15.1.5}$$

so that in compact matrix notation, this becomes

$$\mathbf{AV} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\,. \tag{15.1.6}$$

The matrix $\hat{\mathbf{\Sigma}}$ is an $n \times n$ diagonal matrix with positive entries provided the matrix **A** is of full rank. The matrix $\hat{\mathbf{U}}$ is an $m \times n$ matrix with orthonormal columns, and the matrix $\mathbf{V}$ is an $n \times n$ unitary matrix. Since $\mathbf{V}$ is unitary, the above equation can be solved for $\mathbf{A}$ by multiplying on the right with $\mathbf{V}^*$ so that

$$\mathbf{A} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\mathbf{V}^*\,. \tag{15.1.7}$$

This factorization is known as the *reduced singular value decomposition*, or reduced SVD, of the matrix $\mathbf{A}$. Graphically, the factorization is represented in Fig. 15.3.

The reduced SVD is not the standard definition of the SVD used in the literature. What is typically done to augment the treatment above is to construct a matrix $\mathbf{U}$ from $\hat{\mathbf{U}}$ by adding an additional $m - n$ columns that are orthonormal to the already existing set in $\hat{\mathbf{U}}$. Thus, the matrix $\mathbf{U}$ becomes an $m \times m$ unitary matrix. In order to make this procedure work, an additional $m - n$ rows of zeros is also added to the $\hat{\mathbf{\Sigma}}$ matrix. These "silent" columns of $\mathbf{U}$ and rows of $\mathbf{\Sigma}$ are shown graphically in Fig. 15.4. In performing this procedure, it becomes evident that rank deficient matrices can easily be handled by the SVD decomposition. In particular, instead of $m - n$ silent rows and matrices, there are now simply $m - r$ silent rows and columns added to the decomposition. Thus, the matrix $\mathbf{\Sigma}$ will have r positive diagonal entries, with the remaining $n - r$ being equal to zero.

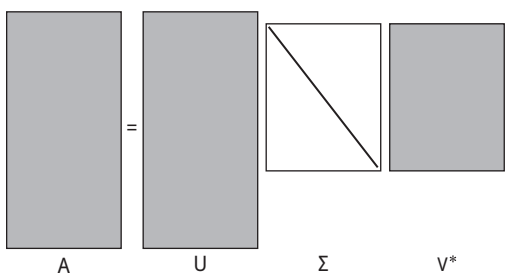


Figure 15.3 Graphical description of the reduced SVD decomposition.

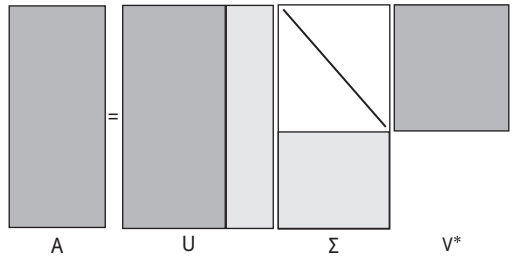


Figure 15.4 Graphical description of the full SVD decomposition, where both $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices. The light-shaded regions of $\mathbf{U}$ and $\mathbf{\Sigma}$ are the silent rows and columns that are extended from the reduced SVD.

The full SVD decomposition thus takes the form

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{15.1.8}$$

with the following three matrices:

$$\mathbf{U} \in \mathbb{C}^{m\times m} \text{ is unitary} \tag{15.1.9}$$

$$\mathbf{V} \in \mathbb{C}^{n\times n} \text{ is unitary} \tag{15.1.10}$$

$$\mathbf{\Sigma} \in \mathbb{R}^{m\times n} \text{ is diagonal.} \tag{15.1.11}$$

Additionally, it is assumed that the diagonal entries of Σ are nonnegative and ordered from largest to smallest so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$, where $p = \min(m, n)$. The SVD decomposition of the matrix $\mathbf{A}$ thus shows that the matrix first applies a unitary transformation preserving the unit sphere via $\mathbf{V}^*$. This is followed by a stretching operation that creates an ellipse with principal semi-axes given by the matrix $\mathbf{\Sigma}$. Finally, the generated hyper-ellipse is rotated by the unitary transformation $\mathbf{U}$. Thus, the statement: the image of a unit sphere under any $m \times n$ matrix is a hyper-ellipse, is shown to be true. The following is the primary theorem concerning the SVD.

Theorem: *Every matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ has a singular value decomposition (15.1.8). Furthermore, the singular values $\{\sigma_j\}$ are uniquely determined, and, if $\mathbf{A}$ is square and the σ_j distinct, the singular vectors $\{\mathbf{u}_j\}$ and $\{\mathbf{v}_j\}$ are uniquely determined up to complex signs (complex scalar factors of absolute value 1).*

15.1.1 Computing the SVD

The above theorem guarantees the existence of the SVD, but in practice, it still remains to be computed. This is a fairly straightforward process if one considers the following matrix products:

$$\begin{aligned}\mathbf{A}^T\mathbf{A} &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*) \\ &= \mathbf{V}\mathbf{\Sigma}\mathbf{U}^*\mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \\ &= \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^* \end{aligned} \tag{15.1.12}$$

and

$$\begin{aligned}\mathbf{A}\mathbf{A}^T &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)^T \\ &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*\mathbf{V}\mathbf{\Sigma}\mathbf{U}^* \\ &= \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^* . \end{aligned} \tag{15.1.13}$$

Multiplying (15.1.12) and (15.1.13) on the right by $\mathbf{V}$ and $\mathbf{U}$, respectively, gives the two self-consistent eigenvalue problems

$$\mathbf{A}^T\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Sigma}^2 \tag{15.1.14}$$

$$\mathbf{A}\mathbf{A}^T\mathbf{U} = \mathbf{U}\mathbf{\Sigma}^2 . \tag{15.1.15}$$

Thus, if the normalized eigenvectors are found for these two equations, then the orthonormal basis vectors are produced for $\mathbf{U}$ and $\mathbf{V}$. Likewise, the square root of the eigenvalues of these equations produces the singular values σ_j.

Example: Consider the SVD decomposition of

$$A = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} . \tag{15.1.16}$$

The following quantities are computed:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix} \tag{15.1.17}$$

$$\mathbf{A}\mathbf{A}^T = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix} . \tag{15.1.18}$$

The eigenvalues are clearly $\lambda = \{9, 4\}$, giving singular values $\sigma_1 = 3$ and $\sigma_2 = 2$. The eigenvectors can similarly be constructed and the matrices $\mathbf{U}$ and $\mathbf{V}$ are given by

$$\begin{bmatrix} \pm 1 & 0 \\ 0 & \pm 1 \end{bmatrix} \tag{15.1.19}$$

where there is an indeterminate sign in the eigenvectors. However, a self-consistent choice of signs must be made. One possible choice gives

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} . \tag{15.1.20}$$

The SVD can be easily computed in Python with the following command:

```
U, S, V = np.linalg.svd(A)
```

where the $[U, S, V]$ correspond to $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$, respectively. This decomposition is a critical tool for analysing many data-driven phenomena. But before proceeding to such applications, the connection of the SVD with standard and well-known techniques is elucidated.

15.2 The SVD in Broader Context

The SVD is an exceptionally important tool in many areas of application. Part of this is due to its many mathematical properties and its guarantee of existence. In what follows, some of its more important theorems and relations to other standard ideas of linear algebra are considered with the aim of setting the mathematical framework for future applications.

15.2.1 Eigenvalues, eigenvectors, and diagonalization

The concept of eigenvalues and eigenvectors is critical to understanding many areas of application. One of the most important areas where it plays a role is in understanding differential equations. Consider, for instance, the system of differential equations:

$$\frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y} \tag{15.2.1}$$

for some vector $\mathbf{y}(t)$ representing a dynamical system of variables and where the matrix $\mathbf{A}$ determines the interaction among these variables. Assuming a solution of the form $\mathbf{y} = \mathbf{x}\exp(\lambda t)$ results in the eigenvalue problem:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}\,. \tag{15.2.2}$$

The question remains: How are the eigenvalues and eigenvectors found? To consider this problem, we rewrite the eigenvalue problem as

$$\mathbf{A}\mathbf{x} - \lambda\mathbf{I}\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}. \tag{15.2.3}$$

Two possibilities now exist.

15.2.1.1 Option I

The determinant of the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is not zero. If this is true, the matrix is *nonsingular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, can be found. The solution to the eigenvalue problem (15.2.2) is then

$$\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{0} \tag{15.2.4}$$

which implies that $\mathbf{x} = \mathbf{0}$. This trivial solution could have easily been guessed. However, it is not relevant as we require nontrivial solutions for $\mathbf{x}$.

15.2.1.2 Option II

The determinant of the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is zero. If this is true, the matrix is *singular* and its inverse, $(\mathbf{A} - \lambda\mathbf{I})^{-1}$, cannot be found. Although there is no longer a guarantee that there is a solution, it is the only scenario which allows for the possibility of $\mathbf{x} \neq 0$. It is this condition that allows for the construction of eigenvalues and eigenvectors. Indeed, we choose the eigenvalues λ so that this condition holds and the matrix is singular.

Another important operation that can be performed with eigenvalues and eigenvectors is the evaluation of

$$\mathbf{A}^M \tag{15.2.5}$$

where M is a large integer. For large matrices $\mathbf{A}$, this operation is computationally expensive. However, knowing the eigenvalues and eigenvectors of $\mathbf{A}$ allows for a significant ease in computational expense. Assuming we have all the eigenvalues and eigenvectors of $\mathbf{A}$, then

$$\begin{aligned}
\mathbf{A}\mathbf{x}_1 &= \lambda_1\mathbf{x}_1 \\
\mathbf{A}\mathbf{x}_2 &= \lambda_2\mathbf{x}_2 \\
&\vdots \\
\mathbf{A}\mathbf{x}_n &= \lambda_1\mathbf{x}_n\,.
\end{aligned}$$

This collection of eigenvalues and eigenvectors gives the matrix system

$$\mathbf{A}\mathbf{S} = \mathbf{S}\mathbf{\Lambda} \tag{15.2.6}$$

where the columns of the matrix $\mathbf{S}$ are the eigenvectors of $\mathbf{A}$,

$$\mathbf{S} = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{pmatrix}, \tag{15.2.7}$$

and $\mathbf{\Lambda}$ is a matrix whose diagonals are the corresponding eigenvalues

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n \end{pmatrix}. \tag{15.2.8}$$

By multiplying (15.2.6) on the right by $\mathbf{S}^{-1}$, the matrix $\mathbf{A}$ can then be rewritten as (note the similarity between this expression and (15.1.8) for the SVD decomposition)

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}\,. \tag{15.2.9}$$

The final observation comes from

$$\mathbf{A}^2 = (\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1})(\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}) = \mathbf{S}\mathbf{\Lambda}^2\mathbf{S}^{-1}\,. \tag{15.2.10}$$

This then generalizes to

$$\mathbf{A}^M = \mathbf{S}\mathbf{\Lambda}^M\mathbf{S}^{-1} \tag{15.2.11}$$

where the matrix $\mathbf{\Lambda}^M$ is easily calculated as

$$\mathbf{\Lambda}^M = \begin{pmatrix} \lambda_1^M & 0 & \cdots & & 0 \\ 0 & \lambda_2^M & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & \lambda_n^M \end{pmatrix}. \tag{15.2.12}$$

Since raising the diagonal terms to the Mth power is easily accomplished, the matrix $\mathbf{A}$ can then be easily calculated by multiplying the three matrices in (15.2.11).

Diagonalization can also recast a given problem so as to elucidate its more fundamental dynamics. A classic example of the use of diagonalization is a two-mass spring system, where the masses m_1 and m_2 are acted on by forces $F_1(t)$ and $F_2(t)$. A schematic of this situation is depicted in Fig. 15.5. For each mass, we can write down Newton's law:

$$\sum F_1 = m_1\frac{d^2x_1}{dt^2} \quad \text{and} \quad \sum F_2 = m_2\frac{d^2x_2}{dt^2} \tag{15.2.13}$$

where $\sum F_1$ and $\sum F_2$ are the sums of the forces on m_1 and m_2, respectively. Note that the equations for $x_1(t)$ and $x_2(t)$ are coupled because of the spring with spring constant k_2. The resulting governing equations are then of the form:

$$m_1\frac{d^2x_1}{dt^2} = -k_1x_1 + k_2(x_2 - x_1) + F_1 = -(k_1 + k_2)x_1 + k_2x_2 + F_1 \tag{15.2.14}$$

$$m_2\frac{d^2x_2}{dt^2} = -k_3x_2 - k_2(x_2 - x_1) + F_2 = -(k_2 + k_3)x_2 + k_2x_1 + F_2. \tag{15.2.15}$$

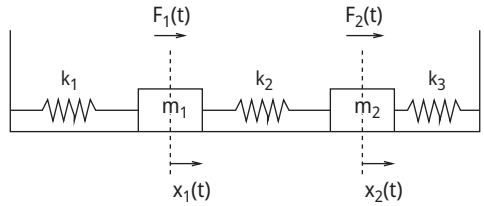


Figure 15.5 A two-mass, three-spring system. The fundamental behaviour of the system can be understood by decomposition of the system via diagonalization. This reveals that all motion can be expressed as the sum of two fundamental motions: the masses oscillating in-phase, and the masses oscillating exactly out-of-phase.

This can be reduced further by assuming, for simplicity, $m = m_1 = m_2$, $K = k_1/m = k_2/m$, and $F_1 = F_2 = 0$. This results in the linear system that can be diagonalized via (15.2.9). The pairs of complex conjugate eigenvalues are produced: $\lambda_1^{\pm} = \pm i(2K + \sqrt{2K})^{1/2}$ and $\lambda_2^{\pm} = \pm i(2K - \sqrt{2K})^{1/2}$. Upon diagonalization, the full system can be understood as simply a linear combination of oscillations of the masses that are in-phase with each other or out-of-phase with each other.

15.2.2 Diagonalization via SVD

Like the eigenvalue and eigenvector diagonalization technique presented earlier, the SVD method also makes the following claim: *the SVD makes it possible for every matrix to be diagonal if the proper bases for the domain and range are used.* To consider this statement more formally, consider that since $\mathbf{U}$ and $\mathbf{V}$ are orthonormal bases in $\mathbb{C}^{m\times m}$ and $\mathbb{C}^{n\times n}$, respectively, then any vector in these spaces can be expanded in their bases. Specifically, consider a vector $\mathbf{b} \in \mathbb{C}^{m\times m}$ and $\mathbf{x} \in \mathbb{C}^{n\times n}$. Then each can be expanded in the bases of $\mathbf{U}$ and $\mathbf{V}$ so that

$$\mathbf{b} = \mathbf{U}\hat{\mathbf{b}}, \quad \mathbf{x} = \mathbf{V}\hat{\mathbf{x}} \tag{15.2.16}$$

where the vectors $\hat{\mathbf{b}}$ and $\hat{\mathbf{x}}$ give the weightings for the orthonormal bases expansion. Now consider the simple equation:

$$\begin{aligned} \mathbf{A}\mathbf{x} = \mathbf{b} \quad \rightarrow \quad & \mathbf{U}^*\mathbf{b} = \mathbf{U}^*\mathbf{A}\mathbf{x} \\ & \mathbf{U}^*\mathbf{b} = \mathbf{U}^*\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*\mathbf{x} \\ & \hat{\mathbf{b}} = \mathbf{\Sigma}\hat{\mathbf{x}}\,. \end{aligned} \tag{15.2.17}$$

Thus, the last line shows that $\mathbf{A}$ reduces to the diagonal matrix $\mathbf{\Sigma}$ when the range is expressed in terms of the basis vectors of $\mathbf{U}$, and the domain is expressed in terms of the basis vectors of $\mathbf{V}$.

Thus, matrices can be diagonalized via either an eigenvalue decomposition or an SVD decomposition. However, there are three key differences in the diagonalization process.

(1) The SVD performs the diagonalization using two different bases, $\mathbf{U}$ and $\mathbf{V}$, while the eigenvalue method uses a single basis $\mathbf{X}$.

(2) The SVD method uses an orthonormal basis while the basis vectors in $\mathbf{X}$, although linearly independent, are not generally orthogonal.

(3) Finally, the SVD is guaranteed to exist for any matrix $\mathbf{A}$, while the same is not true, even for square matrices, for the eigenvalue decomposition.

15.2.3 Useful theorems of SVD

Having established the similarities and connections between eigenvalues and singular values, in what follows a number of important theorems are outlined concerning the SVD. These theorems are important for several reasons. First, the theorems play a key role in the

numerical evaluation of many matrix properties via the SVD. Second, the theorems guarantee certain behaviours of the SVD that can be capitalized upon for future applications. Here is a list of important results. A more detailed account is given by Trefethen and Bau [58].

Theorem: *If the rank of* $\mathbf{A}$ *is* r*, then there are* r *nonzero singular values.*

The proof of this is based upon the fact that the rank of a diagonal matrix is equal to the number of its nonzero entries. And because of the decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$, where $\mathbf{U}$ and $\mathbf{V}$ are full rank, then $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{\Sigma}) = r$. As a side note, the rank of a matrix can be found with the Python command:

```
rank_A = np.linalg.matrix_rank(A)
```

The standard algorithm used to compute the rank is based upon the SVD and the computation of the nonzero singular values. Thus, the theorem is quite useful in practice.

Theorem: *The* range$(\mathbf{A})$=$\langle \mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_r \rangle$ *and* null $(\mathbf{A})$=$\langle \mathbf{v}_{r+1}, \mathbf{v}_{r+2}, \cdots, \mathbf{v}_n \rangle$.

Note that the range and null space come from the two expansion bases $\mathbf{U}$ and $\mathbf{V}$. The range and null space can be found in Python with the commands:

```
range_A = U[:, :rank_A]
null_space_A = V[rank_A:].T
```

This theorem also serves as the most accurate computation basis for determining the range and null space of a given matrix $\mathbf{A}$ via the SVD.

Theorem: *The norm* $\|\mathbf{A}\|_2 = \sigma_1$ *and* $\|\mathbf{A}\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2}$.

These norms are known as the 2-norm and the Frobenius norm, respectively. They essentially measure the *energy* of a matrix. Although it is difficult to conceptualize abstractly what this means, it will become much more clear in the context of given applications. This result is established by the fact that $\mathbf{U}$ and $\mathbf{V}$ are unitary operators so that their norm is unity. Thus, with $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$, the 2-norm is $\|\mathbf{A}\|_2 = \|\mathbf{\Sigma}\|_2 = \max\{|\sigma_j|\} = \sigma_1$. Similar reasoning holds for the Frobenius norm definition. The norm can be calculated in Python with

```
frobenius_norm_A = np.linalg.norm(A, 'fro')
```

Notice that the Frobenius norm contains the total matrix energy, while the 2-norm definition contains the energy of the largest singular value. The ratio $\|\mathbf{A}\|_2/\|\mathbf{A}\|_F$ effectively measures

the portion of the energy in the semi-axis $\mathbf{u}_1$. This fact will be tremendously important for us. Furthermore, this theorem gives the standard way for computing matrix norms.

Theorem: *The nonzero singular values of* $\mathbf{A}$ *are the square roots of the nonzero eigenvalues of* $\mathbf{A}^*\mathbf{A}$ *or* $\mathbf{A}\mathbf{A}^*$. *(These matrices have the same nonzero eigenvalues).*

This has already been shown in the calculation for actually determining the SVD. In particular, this process is illustrated in (15.1.13) and (15.1.12). This theorem is important for actually producing the unitary matrices $\mathbf{U}$ and $\mathbf{V}$.

Theorem: *If* $\mathbf{A} = \mathbf{A}^*$ *(self-adjoint), then the singular values of* $\mathbf{A}$ *are the absolute values of the eigenvalues of* $\mathbf{A}$.

As with most self-adjoint problems, there are very nice properties of the matrices, such as the above theorem. Eigenvalues can be computed with the command:

```
D, V = np.linalg.eig(A)
```

Alternatively, one can specify the number of eigenvalues desired and their specific ordering.

Theorem: *For* $\mathbf{A} \in \mathbb{C}^{m\times m}$, *the determinant is given by* $|\det(A)| = \prod_{j=1}^{m} \sigma_j$.

Again, due to the fact that the matrices $\mathbf{U}$ and $\mathbf{V}$ are unitary, their determinants are unity. Thus, $|\det(A)| = |\det(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)| = |\det(\mathbf{U})||\det(\mathbf{\Sigma})||\det(\mathbf{V}^*)| = |\det(\mathbf{\Sigma})| = \prod_{j=1}^{m} \sigma_j$. Thus, even determinants can be computed via the SVD and its singular values.

15.2.4 Low-dimensional reductions

Now comes the last, and most formative, property associated with the SVD: *low-dimensional approximations* to high degree of freedom or complex systems. In linear algebra terms, this is also *low-rank approximations*. The interpretation of the theorems associated with these low-dimensional reductions are critical for the use and implementation of the SVD. Thus, we consider the following:

Theorem: $\mathbf{A}$ *is the sum of* r *rank-one matrices*

$$\mathbf{A} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \,. \tag{15.2.18}$$

There are a variety of ways to express the $m \times n$ matrix $\mathbf{A}$ as a sum of rank-one matrices. The bottom line is this: *the* N*th partial sum captures as much of the matrix* $\boldsymbol{A}$ *as possible*. Thus, the partial sum of the rank-one matrices is an important object to consider. This leads to the following theorem:

Theorem: *For any N so that $0 \leq N \leq r$, we can define the partial sum*

$$\mathbf{A}_N = \sum_{j=1}^{N} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \,. \tag{15.2.19}$$

And if $N = \min\{m, n\}$, define $\sigma_{N+1} = 0$. Then

$$\|A - A_N\|_2 = \sigma_{N+1} \,. \tag{15.2.20}$$

Likewise, if using the Frobenius norm, then

$$\|A - A_N\|_N = \sqrt{\sigma_{N+1}^2 + \sigma_{N+2}^2 + \cdots + \sigma_r^2} \,. \tag{15.2.21}$$

Interpreting this theorem is critical. Geometrically, we can ask *what is the best approximation of a hyper-ellipsoid by a line segment? Simply take the line segment to be the longest axis, i.e. that associated with the singular value σ_1*. Continuing this idea, what is the best approximation by a two-dimensional ellipse? Take the longest and second longest axes, i.e. those associated with the singular values σ_1 and σ_2. After r steps, the total energy in $\mathbf{A}$ is completely captured. **Thus, the SVD gives a type of least-square fitting algorithm, allowing us to project the matrix onto low-dimensional representations in a formal, algorithmic way.** Herein lies the ultimate power of the method.

15.3 Introduction to Principal Component Analysis (PCA)

To make explicit the concept of the SVD, a simple model example will be formulated that will illustrate all the key concepts associated with the SVD. The model to be considered will be a simple spring–mass system as illustrated in Fig. 15.6. Of course, this is a fairly easy problem to solve from basic concepts of $\mathbf{F} = m\mathbf{a}$. But for the moment, let us suppose we did not know the governing equations. In fact, our aim in this section is to use a number of cameras (probes) to extract out data concerning the behaviour of the system and then to extract empirically the governing equations of motion.

This prologue highlights one of the key applications of the SVD, or alternatively a variant of *principal component analysis* (PCA), *proper mode decomposition* (POD), *Hotelling transform*, *empirical orthogonal functions* (EOF), *reduced order modelling* (ROM), *dimensionality reduction* or *Karhunen–Loéve decomposition*, as it is also known in the literature. Namely, from seemingly complex, perhaps random data, can low-dimensional reductions of the dynamics and behaviour be produced when the governing equations are not known? Such methods can be used to quantify low-dimensional dynamics arising in such areas as turbulent fluid flows [59], structural vibrations [60; 61], insect locomotion [62], damage detection [63], and neural decision-making strategies [64], to name just a few areas of application. It will also become obvious as we move forward on this topic that the breadth of applications is staggering

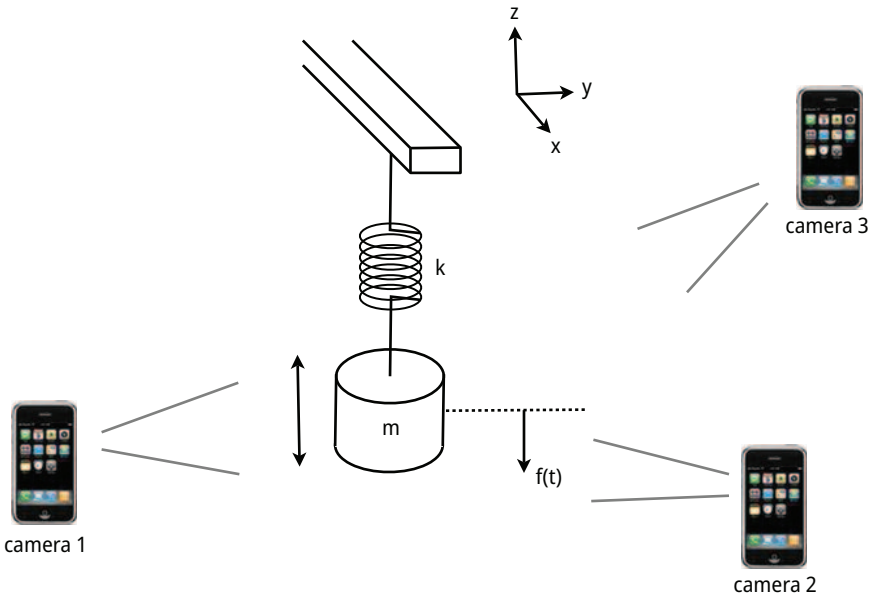


Figure 15.6 A prototypical example of how we might apply a principal component analysis, or SVD, is the simple mass–spring system exhibited here. The mass m is suspended with a spring with Hook's constant k. Three video cameras collect data about its motion in order to ascertain its governing equations.

and includes image processing and signal analysis. Thus, the perspective to be taken here is clearly one in which the data analysis of an unknown, but potentially low-dimensional system is to be analysed.

Again we turn our attention to the simple experiment at hand: a mass suspended by a spring as depicted in Fig. 15.6. If the mass is perturbed or taken from equilibrium in the z-direction only, we know that the governing equations are simply

$$\frac{d^2 f(t)}{dt^2} = -\omega^2 f(t) \tag{15.3.1}$$

where the function $f(t)$ measures the displacement of the mass in the z-direction as a function of time. This has the well-known solution (in amplitude–phase form)

$$f(t) = A\cos(\omega t + \omega_0) \tag{15.3.2}$$

where the values of A and ω_0 are determined from the initial state of the system. This essentially states that the state of the system can be described by a one-degree-of-freedom system.

In the above analysis, there are many things we have ignored. Included in the list of things we have ignored is the possibility that the initial excitation of the system actually produces movement in the $x-y$ plane. Further, there is potentially noise in the data from, for instance,

shaking of the cameras during the video capture. Moreover, from what we know of the solution, only a single camera is necessary to capture the underlying motion. In particular, a single camera in the $x-y$ plane at $z = 0$ would be ideal. Thus, we have oversampled the data with three cameras and have produced redundant data sets. From all of these potential perturbations and problems, it is our goal to extract out the simple solution given by the simple harmonic motion.

This problem is a good example of what kind of processing is required to analyse a realistic data set. Indeed, one can imagine that most data will be quite noisy, perhaps redundant, and certainly not produced from an optimal viewpoint. But through the process of PCA, these can be circumvented in order to extract out the ideal or simplified behaviour. Moreover, we may even learn how to transform the data into the optimal viewpoint for analysing the data.

15.3.1 Data collection and ordering

Assume now that we have started the mass in motion by applying a small perturbation in the z-direction only. Thus, the mass will begin to oscillate. Three cameras are then used to record the motion. Each camera produces a two-dimensional representation of the data. If we denote the data from the three cameras with subscripts a, b, and c, then the data collected are represented by the following:

$$\text{camera 1: } (\mathbf{x}_a, \mathbf{y}_a) \tag{15.3.3}$$

$$\text{camera 2: } (\mathbf{x}_b, \mathbf{y}_b) \tag{15.3.4}$$

$$\text{camera 3: } (\mathbf{x}_c, \mathbf{y}_c) \tag{15.3.5}$$

where each set $(\mathbf{x}_j, \mathbf{y}_j)$ is data collected over time of the position in the $x-y$ plane of the camera. Note that this is not the same $x-y$ plane of the oscillating mass system as shown in Fig. 15.6. Indeed, we should pretend we do not know the correct $x-y-z$ coordinates of the system. Thus, the camera positions and their relative $x-y$ planes are arbitrary. The length of each vector $\mathbf{x}_i$ and $\mathbf{y}_i$ depends on the data collection rate and the length of time the dynamics is observed. We denote the length of these vectors as n.

All the data collected can then be gathered into a single matrix:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{y}_a \\ \mathbf{x}_b \\ \mathbf{y}_b \\ \mathbf{x}_c \\ \mathbf{y}_c \end{bmatrix}. \tag{15.3.6}$$

Thus, the matrix $X \in \mathbb{R}^{m \times n}$, where m represents the number of measurement types and n is the number of data points taken from the camera over time.

Now that the data have been arranged, two issues must be addressed: noise and redundancy. Everyone has an intuitive concept that noise in your data can only deteriorate, or corrupt, your ability to extract the true dynamics. Just as in image processing, noise can alter an image beyond restoration. Thus, there is also some idea that if the measured data are too noisy, fidelity of the underlying dynamics is compromised from a data analysis point of view. The key measure of this is the so-called signal-to-noise ratio: SNR $= \sigma^2_{\text{signal}}/\sigma^2_{\text{noise}}$, where the ratio is given as the ratio of the variances of the signal and noise fields. A high SNR (much greater than unity) gives almost noiseless (high precision) data, whereas a low SNR suggests the underlying signal is corrupted by the noise. The second issue to consider is redundancy. In the example of Fig. 15.6, the single degree of freedom is sampled by three cameras, each of which is really recording the same single degree of freedom. Thus, the measurements should be rife with redundancy, suggesting that the different measurements are statistically dependent. Removing this redundancy is critical for data analysis.

15.3.2 The covariance matrix

An easy way to identify redundant data is by considering the covariance between data sets. Recall from our early chapters on probability and statistics that the covariance measures the statistical dependence/independence between two variables. Obviously, strongly statistically dependent variables can be considered as redundant observations of the system. Specifically, consider two sets of measurements with zero means expressed in row vector form:

$$\mathbf{a} = [a_1\ a_2\ \cdots\ a_n] \quad \text{and} \quad \mathbf{b} = [b_1\ b_2\ \cdots\ b_n] \tag{15.3.7}$$

where the subscript denotes the sample number. The variances of $\mathbf{a}$ and $\mathbf{b}$ are given by

$$\sigma^2_{\mathbf{a}} = \frac{1}{n-1}\mathbf{a}\mathbf{a}^T \tag{15.3.8}$$

$$\sigma^2_{\mathbf{b}} = \frac{1}{n-1}\mathbf{b}\mathbf{b}^T \tag{15.3.9}$$

while the covariance between these two data sets is given by

$$\sigma^2_{\mathbf{ab}} = \frac{1}{n-1}\mathbf{a}\mathbf{b}^T \tag{15.3.10}$$

where the normalization constant of $1/(n-1)$ is for an unbiased estimator.

We do not just have two vectors, but potentially quite a number of experiments and data that would need to be correlated and checked for redundancy. In fact, the matrix in (15.3.6) is exactly what needs to be checked for covariance. The appropriate *covariance matrix* for this case is then

$$\mathbf{C}_{\mathbf{X}} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T\,. \tag{15.3.11}$$

This is easily computed with Python from the command line:

```
covariance_A = np.cov(A, rowvar=False)
```

The covariance matrix $\mathbf{C_X}$ is a square, symmetric $m \times m$ matrix whose diagonal represents the variance of particular measurements. The off-diagonal terms are the covariances between measurement types. Thus, $\mathbf{C_X}$ captures the correlations between all possible pairs of measurements. Redundancy is thus easily captured since if two data sets are identical (identically redundant), the off-diagonal term and diagonal term would be equal since $\sigma^2_{\mathbf{ab}} = \sigma^2_{\mathbf{a}} = \sigma^2_{\mathbf{b}}$ if $\mathbf{a} = \mathbf{b}$. Thus, large off-diagonal terms correspond to redundancy while small off-diagonal terms suggest that the two measured quantities are close to being statistically independent and have low redundancy. It should also be noted that large diagonal terms, or those with large variances, typically represent what we might consider *the dynamics of interest* since the large variance suggests strong fluctuations in that variable. Thus, the covariance matrix is the key component to understanding the entire data analysis.

15.3.2.1 Achieving the goal

The insight given by the covariance matrix leads to our ultimate aim of
Achieving the goal: The insight given by the covariance matrix leads to our ultimate aim of

(1) removing redundancy
(2) identifying those signals with maximal variance.

Thus, in a mathematical sense we are simply asking to represent $\mathbf{C_X}$ so that the diagonals are ordered from largest to smallest and the off-diagonals are zero, i.e. our task is to diagonalize the covariance matrix. This is *exactly* what the SVD does, thus allowing it to become the tool of choice for data analysis and dimensional reduction. In fact, the SVD diagonalizes and each singular direction captures as much energy as possible as measured by the singular values σ_j.

15.4 Principal Components, Diagonalization, and SVD

The example presented in the previous section shows that the key to analysing a given experiment is to consider the covariance matrix

$$\mathbf{C_X} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T \tag{15.4.1}$$

where the matrix $\mathbf{X}$ contains the experimental data of the system. In particular, $\mathbf{X} \in \mathbb{C}^{m\times n}$, where m is the number of probes or measuring positions, and n is the number of experimental data points taken at each location.

In this setup, the following facts are highlighted:

(1) $\mathbf{C_X}$ is a square, symmetric $m \times m$ matrix.

(2) The diagonal terms of $\mathbf{C_X}$ are the variances for particular measurements. By assumption, large variances correspond to *dynamics of interest*, whereas low variances are assumed to correspond to *uninteresting dynamics*.

(3) The off-diagonal terms of $\mathbf{C_X}$ are the covariances between measurements. Indeed, the off-diagonals capture the correlations between all possible pairs of measurements. A large off-diagonal term represents two events that have a high degree of redundancy, whereas a small off-diagonal coefficient means there is little redundancy in the data, i.e. they are statistically independent.

15.4.1 Diagonalization

The concept of diagonalization is critical for understanding the underpinnings of many physical systems. In this process of diagonalization, the correct coordinates, or basis functions, are revealed that reduce the given system to its low-dimensional essence. There is more than one way to diagonalize a matrix, and this is certainly true here as well since the constructed covariance matrix $\mathbf{C_X}$ is square and symmetric, both properties that are especially beneficial for standard eigenvalue/eigenvector expansion techniques.

The key idea behind the diagonalization is simply this: there exists an *ideal* basis in which the $\mathbf{C_X}$ can be written (diagonalized) so that in this basis, all redundancies have been removed, and the largest variances of particular measurements are ordered. In the language being developed here, this means that the system has been written in terms of its *principal components*, or in a *proper orthogonal decomposition*.

15.4.1.1 Eigenvectors and eigenvalues

The most straightforward way to diagonalize the covariance matrix is by making the observation that $\mathbf{XX}^T$ is a square, symmetric $m \times m$ matrix, i.e. it is self-adjoint so that the m eigenvalues are real and distinct. Linear algebra provides theorems which state that such a matrix can be rewritten as

$$\mathbf{XX}^T = \mathbf{S\Lambda S}^{-1} \tag{15.4.2}$$

as stated in (15.2.9), where the matrix $\mathbf{S}$ is a matrix of the eigenvectors of $\mathbf{XX}^T$ arranged in columns. Since it is a symmetric matrix, these eigenvector columns are orthogonal so that ultimately the $\mathbf{S}$ can be written as a unitary matrix with $\mathbf{S}^{-1} = \mathbf{S}^T$. Recall that the matrix $\mathbf{\Lambda}$ is a diagonal matrix whose entries correspond to the m distinct eigenvalues of $\mathbf{XX}^T$.

This suggests that instead of working directly with the matrix $\mathbf{X}$, we consider working with the transformed variable, or in the principal component basis,

$$\mathbf{Y} = \mathbf{S}^T\mathbf{X}\,. \tag{15.4.3}$$

For this new basis, we can then consider its covariance

$$\begin{aligned}\mathbf{C_Y} &= \frac{1}{n-1}\mathbf{Y}\mathbf{Y}^T \\ &= \frac{1}{n-1}(\mathbf{S}^T\mathbf{X})(\mathbf{S}^T\mathbf{X})^T \\ &= \frac{1}{n-1}\mathbf{S}^T(\mathbf{X}\mathbf{X}^T)\mathbf{S} \\ &= \frac{1}{n-1}\mathbf{S}^T\mathbf{S}\mathbf{\Lambda}\mathbf{S}\mathbf{S}^T \\ \mathbf{C_Y} &= \frac{1}{n-1}\mathbf{\Lambda}\,. \end{aligned} \tag{15.4.4}$$

In this basis, the *principal components* are the eigenvectors of $\mathbf{X}\mathbf{X}^T$ with the interpretation that the jth diagonal value of $\mathbf{C_Y}$ is the variance of $\mathbf{X}$ along $\mathbf{x}_j$, the jth column of $\mathbf{S}$. The following lines of code produce the principal components of interest.

```
X = np.random.rand(10,10)

m, n = X.shape
mn = np.mean(X, axis=1)
X = X - mn[:, np.newaxis]

Cx = (1 / (n - 1)) * np.dot(X, X.T)
D, V = np.linalg.eig(Cx)
lambda_vals = np.diag(D)

m_arrange = np.argsort(-lambda_vals) # Sort in decreasing order
  lambda_vals = lambda_vals[m_arrange]
V = V[:, m_arrange]
Y = np.dot(V.T, X) # compute projection
```

This simple code thus produces the eigenvalue decomposition and the projection of the original data onto the principal component basis.

15.4.1.2 Singular value decomposition

A second method for diagonalizing the covariance matrix is the SVD method. In this case, the SVD can diagonalize any matrix by working in the appropriate pair of bases $\mathbf{U}$ and $\mathbf{V}$ as outlined in the first part of this chapter. Thus, by defining the transformed variable

$$\mathbf{Y} = \mathbf{U}^*\mathbf{X} \tag{15.4.5}$$

where $\mathbf{U}$ is the unitary transformation associated with the SVD: $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$. Just as in the eigenvalue/eigenvector formulation, we then compute the variance in $\mathbf{Y}$:

$$\begin{aligned}
\mathbf{C}_\mathbf{Y} &= \frac{1}{n-1}\mathbf{Y}\mathbf{Y}^T \\
&= \frac{1}{n-1}(\mathbf{U}^*\mathbf{X})(\mathbf{U}^*\mathbf{X})^T \\
&= \frac{1}{n-1}\mathbf{U}^*(\mathbf{X}\mathbf{X}^T)\mathbf{U} \\
&= \frac{1}{n-1}\mathbf{U}^*\mathbf{U}\mathbf{\Sigma}^2\mathbf{U}\mathbf{U}^* \\
\mathbf{C}_\mathbf{Y} &= \frac{1}{n-1}\mathbf{\Sigma}^2 . \qquad (15.4.6)
\end{aligned}$$

This makes explicit the connection between the SVD and the eigenvalue method, namely that $\mathbf{\Sigma}^2 = \mathbf{\Lambda}$. The following lines of code produce the principal components of interest using the SVD (assume that you have the first three lines from the previous Python code).

```
u, s, vh = np.linalg.svd(X.T / np.sqrt(n - 1))
lambda_vals = np.diag(s) ** 2
Y = np.dot(u.T, X)
```

This gives the SVD method for producing the principal components. Overall, the SVD method is the more robust method and should be used. However, the connection between the two methods becomes apparent in these calculations.

15.4.2 Spring experiment

To illustrate this completely in practice, three experiments will be performed with the configuration of Fig. 15.6. The following experiments will attempt to illustrate various aspects of the PCA and its practical usefulness and the effects of noise on the PCA algorithms.

(1) **Ideal case** Consider a small displacement of the mass in the z-direction and the ensuing oscillations. In this case, the entire motion is in the z-direction with simple harmonic motion being observed.

(2) **Noisy case** Repeat the ideal case experiment, but this time, introduce camera shake into the video recording. This should make it more difficult to extract the simple harmonic motion. But if the shake is not too bad, the dynamics will still be extracted with the PCA algorithms.

(3) **Horizontal displacement** In this case, the mass is released off-centre so as to produce motion in the $x-y$ plane as well as the z-direction. Thus, there is both a pendulum motion and a simple harmonic oscillation. See what the PCA tells us about the system.

Frames must be extracted out of the mass movement from the video frames, which can be done in various ways in Python. Once the frames are extracted, they can again be converted to double precision numbers for mathematical processing. In this case, the position of the mass is to be determined from each frame. This basic shell of code is enough to begin the process of extracting the spring–mass system information.

15.5 Principal Components and Proper Orthogonal Modes

Now that the basic framework of the PCA and its relation to the SVD has been laid down, a few remaining issues need to be addressed. In particular, the PCA seems to suggest that we are simply expanding our solution in another *orthonormal* basis, one which can always diagonalize the underlying system.

Mathematically, we can consider a given function $f(x,t)$ over a domain of interest. In most applications, the variables x and t will refer to the standard space–time variables we are familiar with. The idea is to then expand this function in some basis representation so that

$$f(x,t) \approx \sum_{j=1}^{N} a_j(t)\phi_j(x) \tag{15.5.1}$$

where N is the total number of modes, $\phi_j(x)$, to be used. The remaining function $a_j(t)$ determines the weights of the spatial modes.

The expansion (15.5.1) is certainly not a new idea to us. Indeed, we have been using this concept extensively already with Fourier transforms, for instance. Specifically, here are some of the more common expansion bases used in practice:

$$\phi_j(x) = (x - x_0)^j \quad \text{Taylor expansion} \tag{15.5.2}$$

$$\phi_j(x) = \cos(jx) \quad \text{Discrete cosine transform} \tag{15.5.3}$$

$$\phi_j(x) = \sin(jx) \quad \text{Discrete sine transform} \tag{15.5.4}$$

$$\phi_j(x) = \exp(jx) \quad \text{Fourier transform} \tag{15.5.5}$$

$$\phi_j(x) = \psi_{a,b}(x) \quad \text{Wavelet transform} \tag{15.5.6}$$

$$\phi_j(x) = \phi_{\lambda_j}(x) \quad \text{Eigenfunction expansion} \tag{15.5.7}$$

where $\phi_{\lambda_j}(x)$ are eigenfunctions associated with the underlying system. This places the concept of a basis function expansion in familiar territory. Further, it shows that such a basis function expansion is not unique, but rather can potentially have an infinite number of different possibilities.

As for the weighting coefficients $a_j(t)$, they are simply determined from the standard inner product rules and the fact that the basis functions are orthonormal (note that they are not written this way above):

$$\int \phi_j(x)\phi_n(x)dx = \begin{cases} 1 & j = n \\ 0 & j \neq n. \end{cases} \tag{15.5.8}$$

This then gives for the coefficients

$$a_j(t) = \int f(x,t)\phi_j(x)dx \tag{15.5.9}$$

and the basis expansion is completed.

Interestingly enough, the basis functions selected are often chosen for simplicity and/or their intuitive meaning for the system. For instance, the Fourier transform very clearly highlights the fact that the given function has a representation in terms of *frequency* components. This is fundamental for understanding many physical problems, as there is clear and intuitive meaning to the Fourier modes.

In contrast to selecting basis functions for simplicity or intuitive meaning, the broader question can be asked: what criteria should be used for selecting the functions $\phi_j(x)$? This is an interesting question given that any complete basis can approximate the function $f(x,t)$ to any desired accuracy given N large enough. But what is desired is the best basis functions such that, with N as small as possible, we achieve the desired accuracy. The goal is then the following: find a sequence of orthonormal basis functions $\phi_j(x)$ so that the first two terms give the best two-term approximation to $f(x,t)$, or the first five terms give the best five-term approximation to $f(x,t)$. These special, ordered, orthonormal functions are called the *proper orthogonal modes* for the function $f(x,t)$. With these modes, the expansion (15.5.9) is called the POD of $f(x,t)$. The relation to the SVD will become obvious momentarily.

Example 1: Consider an approximation to the following surface:

$$f(x,t) = e^{-|(x-0.5)(t-1)|} + \sin(xt) \quad x \in [0,1],\ t \in [0,2]. \tag{15.5.10}$$

As listed above, there are a number of methods available for approximating this function using various basis functions. And all of the ones listed are guaranteed to converge to the surface for a large enough N in (15.5.1).

In the POD method, the surface is first discretized and approximated by a finite number of points. In particular, the following discretization will be used:

```
x = np.linspace(0, 1, 25)
t = np.linspace(0, 2, 50)
```

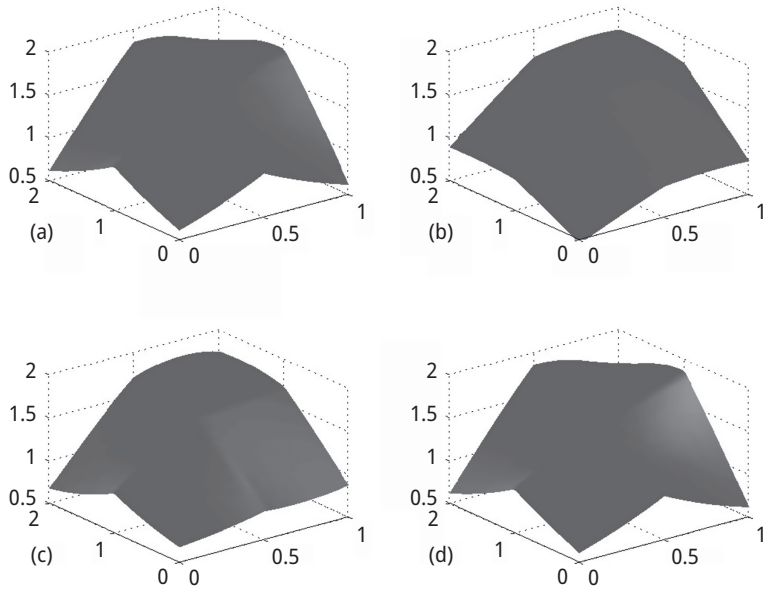


Figure 15.7 Representation of the surface (a) by a series of low-rank (low-dimensional) approximations with (b) one mode, (c) two modes, and (d) three modes. The energy captured in the first mode is approximately 92% of the total surface energy. The first three modes together capture 99.9% of the total surface energy, thus suggesting that the surface can be easily and accurately represented by a three-mode expansion.

where x has been discretized into 25 points and t is discretized into 50 points. The surface is represented in Fig. 15.7(a) over the domain of interest. The surface is constructed by using the **meshgrid** command as follows:

```
T, X = np.meshgrid(t, x)
f = np.exp(-np.abs((X - 0.5) * (T - 1))) + np.sin(X * T)
fig = plt.figure(figsize=(10, 6))
ax1 = fig.add_subplot(2, 2, 1, projection='3d')
ax1.plot_surface(X, T, f, color='gray')
```

The **surfl** produces a lighted surface that in this case is represented in grey-scale. Note that in producing this surface, the matrix **f** is a 25×50 matrix so that the x-values are given as row locations, while the t-values are given by column locations.

The basis functions are then computed using the SVD. Recall that the SVD produces ordered singular values and associated orthonormal basis functions that capture as much energy as possible. Thus, an SVD can be performed on the matrix **f** that represents the surface.

```
u, s, vh = np.linalg.svd(f)

for j in range(3):
    ff = np.dot(u[:, :j+1], np.dot(np.diag(s[:j+1]), vh[:j+1, :]))
    ax = fig.add_subplot(2, 2, j+2, projection='3d')
    ax.plot_surface(X, T, ff, color='gray')
    ax.set_zlim(0.5, 2)
```

The SVD command pulls out the diagonal matrix along with the two unitary matrices $\mathbf{U}$ and $\mathbf{V}$. The loop above processes the sum of the first, second, and third modes. This gives the POD modes of interest. Figure 15.7 demonstrates the modal decomposition and the accuracy achieved with representing the surface with one, two, and three POD modes. The first mode alone captures 92% of the surface, while three modes produce a staggering 99.9% of the energy. This should be contrasted with Fourier methods, for instance, which would require potentially hundreds of modes to achieve such accuracy. *As stated previously, these are the best one, two, and three mode approximations of the function* $f(x,t)$ *that can be achieved.*

To be more precise about the nature of the POD, consider the *energy* in each mode. This can be easily computed, or has already been computed, in the SVD, i.e. they are the singular values σ_j. In Python, the energy in the one-mode and three-mode approximation is computed from

```
sig = np.diag(s)
energy1 = sig[0] / np.sum(sig)
energy3 = np.sum(sig[:3]) / np.sum(sig)
```

More generally, the entire *spectrum* of singular values can be plotted. Figure 15.8 shows the complete spectrum of singular values on a standard plot and a log plot. The clear dominance of a single mode is easily deduced. Additionally, the first three POD modes are illustrated: they are the first three columns of the matrix $\mathbf{U}$. These constitute the orthonormal expansion basis of interest. Note that the key part of the code is simply the calculation of the POD modes which are weighted, in practice, by the singular values and their evolution in time as given by the columns of **V**.

Example 2: Consider the following time–space function:

$$f(x,t) = [1 - 0.5\cos 2t]\mathrm{sech}x + [1 - 0.5\sin 2t]\mathrm{sech}x\tanh x\,. \tag{15.5.11}$$

This represents an asymmetric, time-periodic breather of a localized solution. Such solutions arise in a myriad of contexts and are often obtained via full numerical simulations of an underlying physical system. Let us once again apply the techniques of POD analysis to see

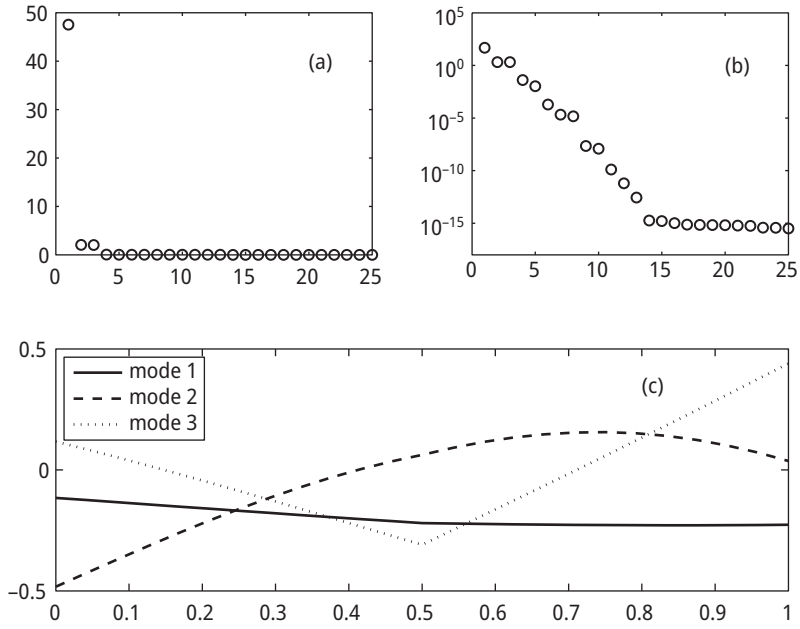


Figure 15.8 Singular values on a standard plot (a) and a log plot (b) showing the energy in each POD mode. For the surface considered in Fig. 15.7, the bulk of the energy is in the first POD mode. A three-mode approximation produces 99.9% of the energy. The linear POD modes are illustrated in panel (c).

what is involved in the dynamics of this system. A discretization of the system is first applied and made into a two-dimensional representation in time and space.

```
x = np.linspace(-10, 10, 100)
t = np.linspace(0, 10, 30)
X, T = np.meshgrid(x, t)
f = np.cosh(X) ** -1 * (1 - 0.5 * np.cos(2 * T)) +
     (np.cosh(X) ** -1 * np.tanh(X)) * (1 - 0.5 * np.sin(2 * T))
```

Note that, unlike before, the matrix **f** is now a 30×100 matrix so that the x-values are along the columns and t-values are along the rows. This is done simply so that we can use the **waterfall** command in Python.

The SVD of the matrix **f** produces the quantities of interest. In this case, since we want the x-value in the rows, the transpose of the matrix is considered in the SVD.

```
u, s, v = np.linalg.svd(f.T)
for j in range(3):
    ff = np.dot(u[:, :j+1], np.dot(np.diag(s[:j+1]), v[:j+1, :]))
    ax = fig.add_subplot(2, 2, j+2, projection='3d')
    ax.plot_surface(X, T, ff.T, color='gray')
```

This produces the original function along with the first three modal approximations of the function. As is shown in Fig. 15.9, the two-mode and three-mode approximations appear to be identical. In fact, they are. The singular values of the SVD show that ≈83% of the energy is in the first mode of Fig. 15.9(a), while 100% of the energy is captured by a two-mode approximation as shown in Fig. 15.9(b). Thus, the third mode is completely unnecessary. As before, we can also produce the first two modes, which are the first two columns of $\mathbf{U}$, along with their time behaviour, which are the first two columns of $\mathbf{V}$.

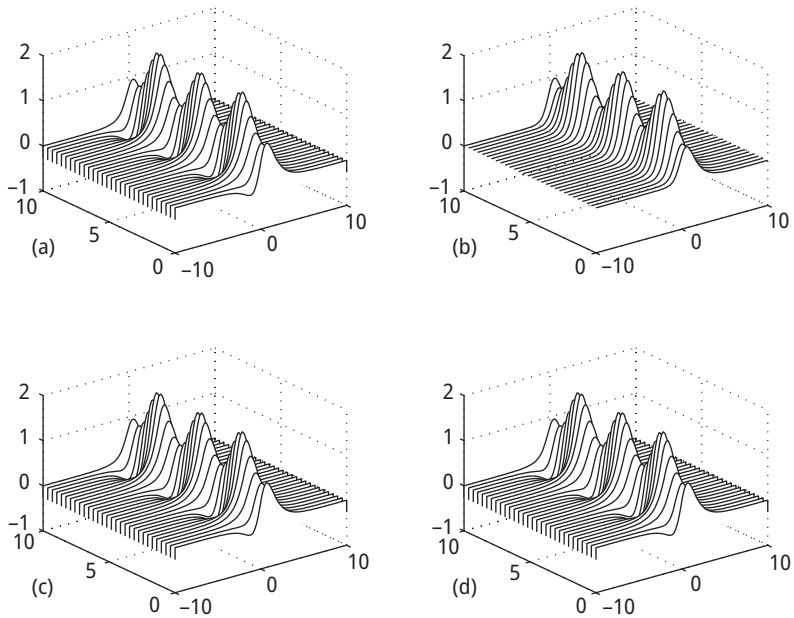


Figure 15.9 Representation of the time–space dynamics (a) by a series of low-rank (low-dimensional) approximations with (b) one mode, (c) two modes, and (d) three modes. The energy captured in the first mode is approximately 83% of the total energy. The first two modes together capture 100% of the surface energy, thus suggesting that the surface can be easily and accurately represented by a simple two-mode expansion.

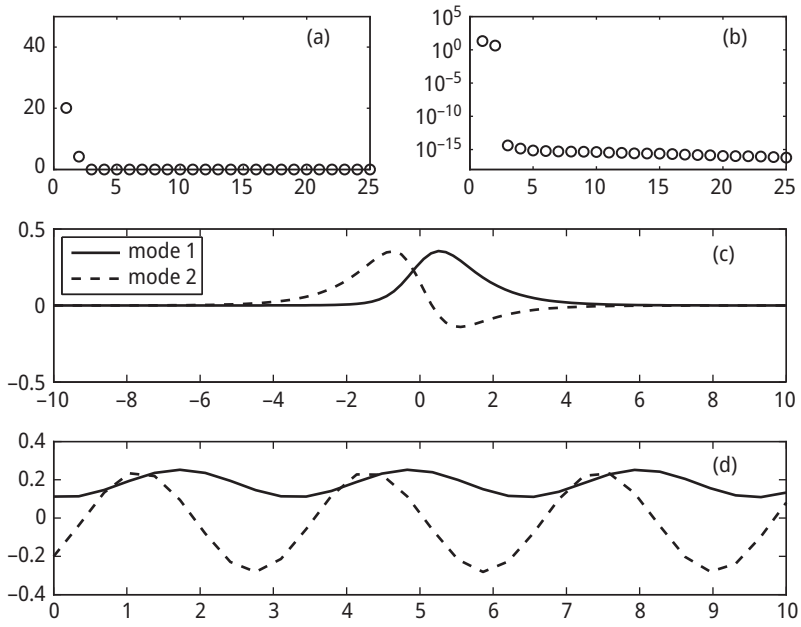


Figure 15.10 Singular values on a standard plot (a) and a log plot (b) showing the energy in each POD mode. For the space–time surface considered in Fig. 15.9, the bulk of the energy is in the first POD mode. A two-mode approximation produces 100% of the energy. The linear POD modes are illustrated in panel (c) which their time evolution behaviour in panel (d).

Figure 15.10 shows the singular values along with the spatial and temporal behaviour of the first two modes. It is not surprising that the SVD characterizes the total behaviour of the system as an exactly (to numerical precision) two-mode behaviour. Recall that is exactly what we started with! Note, however, that our original two modes are different from the SVD modes. Indeed, the first SVD mode is asymmetric unlike our symmetric hyperbolic secant.

15.5.1 Summary and limits of SVD/PCA/POD

The methods of PCA and POD, which are essentially related to the idea of diagonalization of any matrix via the SVD, are exceptionally powerful tools and methods for the evaluation of data-driven systems. Further, they provide the most precise method for performing low-dimensional reductions of a given system. A number of key steps are involved in applying the method to experimental or computational data sets.

(1) Organize the data into a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$, where m is the number of measurement types (or probes) and n is the number of measurements (or trials) taken from each probe.

(2) Subtract off the mean for each measurement type or row $\mathbf{x}_j$.

(3) Compute the SVD and singular values of the covariance matrix to determine the principal components.

If considering a fitting algorithm, then the POD method is the appropriate technique to consider and the SVD can be applied directly to the $\mathbf{A} \in \mathbb{C}^{m\times n}$ matrix. This then produces the singular values as well as the POD modes of interest.

Although extremely powerful, and seemingly magical in its ability to produce insightful quantification of a given problem, these SVD methods are not without limits. Some fundamental assumptions have been made in producing the results thus far, the most important being the assumption of *linearity*. Recently some efforts to extend the method using nonlinearity prior to the SVD have been explored [65–67]. A second assumption is that larger variances (singular values) represent more important dynamics. Although generally this is believed to be true, there are certainly cases where this assumption can be quite misleading. Additionally, in constructing the covariance matrix, it is assumed that the mean and variance are sufficient statistics for capturing the underlying dynamics. It should be noted that only exponential (Gaussian) probability distributions are completely characterized by the first two moments of the data. This will be considered further in Chapter 16 on independent component analysis. Finally, it is often the case that PCA may not produce the optimal results. For instance, consider a point on a propeller blade. Viewed from the PCA framework, two orthogonal components are produced to describe the circular motion. However, this is really a one-dimensional system that is solely determined by the phase variable. The PCA misses this fact unless use is made of the information already known, i.e. that a polar coordinate system with fixed radius is the appropriate context to frame the problem. Thus, blindly applying the technique can also lead to nonoptimal results that miss obvious facts.

15.6 Robust PCA

The SVD and its various guises, most notably PCA, are perhaps the most widely used statistical tools for generating dimensionality reduction of data. As has been demonstrated and discussed, the SVD produces characteristic features (principal components) that are determined by the covariance matrix of the data. Fundamental in producing the SVD/PCA/POD modes is the L^2 norm for data fitting. Although the L^2 norm is highly appealing due to its centrality (and physical interpretation) in most applications, it does have potential flaws that can severely limit its applicability. Specifically, if corrupt data or large noise fluctuations are present in the data matrix, the higher-dimensional least-square fitting performed by the SVD algorithm squares this pernicious error, leading to significant deformation of the singular values and PCA modes describing the data. Although many physical systems and/or computations are relatively free of data corruption, many modern applications in image processing, PIV fluid measurements, web data analysis, and bioinformatics, for instance, are rife with arbitrary corruption of the measured data, thus ensuring that standard application of the SVD will be of limited use.

Ideally, in performing dimensionality reduction one should not allow the corrupt or large noise fluctuations to so grossly influence one's results. In order to limit the impact of such *data outliers*, a different measure, or norm, can be envisioned in measuring the best data fit (SVD/PCA/POD modes). One measure that has recently produced great success in this arena is the L^1 norm [68], which no longer squares the distance of the data to the best-fit mode. Indeed, the L^1 norm has been demonstrated to promote sparsity and is the underlying theoretical construct in *compressive sensing* or *sparse sensing* algorithms. Here, the L^1 norm is simply used as an alternative measure (norm) for performing the equivalent of the least-square fit to the data. As a result, a more robust method is achieved of computing PCA components, i.e. the so-called *robust PCA* method.

15.6.1 Matrix decomposition: Low-rank plus sparse

To illustrate the entanglement of low-rank data with corrupt (sparse) measurement/matrix error, we can construct a new matrix, which is composed of these two elements together

$$\mathbf{M} = \mathbf{L} + \mathbf{S} \tag{15.6.1}$$

where $\mathbf{M}$ is the data matrix of interest that is composed of a low-rank structure $\mathbf{L}$ along with some sparse data $\mathbf{S}$. Our objective is the following: given observations $\mathbf{M}$, can the low-rank matrix $\mathbf{L}$ and sparse matrix $\mathbf{S}$ be recovered? Adding to the difficulty of this task is the following: we do not know the rank of the matrix $\mathbf{L}$, nor do we know how many nonzero (sparse) elements of the matrix $\mathbf{S}$ exist.

Remarkably, Candés and co-workers [68] have recently proved that this can indeed be solved through the formulation of a convex optimization problem. At first sight, the separation problem seems impossible to solve since the number of unknowns to infer for $\mathbf{L}$ and $\mathbf{S}$ is twice as many as the given measurements in $\mathbf{M}$. Furthermore, it seems even more daunting that we expect to reliably obtain the low-rank matrix $\mathbf{L}$ with errors in $\mathbf{S}$ of arbitrarily large magnitude. But not only can this problem be solved, it can be solved by tractable convex optimization [68].

Of course, in real applications, it is rare that the matrix decomposition is as ideal as (15.6.1). More generally, the decomposition is of the form

$$\mathbf{M} = \mathbf{L} + \mathbf{S} + \mathbf{N} \tag{15.6.2}$$

where the matrix $\mathbf{N}$ is a dense, small perturbation accounting for the fact that the low-rank component is only approximately low-rank and that small errors can be added to all the entries (in some sense, this model unifies the classical PCA and the robust PCA by combining both sparse gross errors and dense small noise). But even in this case, the convex optimization algorithm formulated by Candés and co-workers [68] seems to do a remarkable job at separating low-rank from sparse data. This can be used to great advantage when certain types of data are considered, namely those with corrupt data or large, sparse noisy perturbations.

The details of the method and its proof are beyond the scope of this book. However, we will utilize the algorithms developed in the robust PCA literature in order to separate data matrices into low-rank and sparse components. A number of Python algorithms can be downloaded from the web for this purpose (http://perception.csl.illinois.edu/matrix-rank/sample_code.html). Indeed, a wide variety of techniques have been developed for specifically providing a framework for robust PCA, including the augmented Lagrange multiplier (ALM) method [69], accelerated proximal gradient method [70], dual method [70], singular value thresholding method [71], and the alternating direction method [72]. For our purposes, we will use the Python program **inexact_alm_rpca** since it is extremely simple to use and exceedingly fast [69].

Example: As an example, consider the function from the previous section

$$f(x,t) = [1 - 0.5\cos 2t]\mathrm{sech}x + [1 - 0.5\sin 2t]\mathrm{sech}x\tanh x\,. \tag{15.6.3}$$

Previously, the SVD was applied to this function and a two-mode dominance was clearly demonstrated (see Figs. 15.9 and 15.10). To illustrate the problem to be considered, the above function is plotted along with a corrupted version in Fig. 15.11. In this example, the data were corrupted by the addition of sparse, but large, noise fluctuations to the ideal data. The following Python code produces the ideal figure.

```
n = 200
x = np.linspace(-10, 10, n)
t = np.linspace(0, 10, 30)
X, T = np.meshgrid(x, t)
usol = np.cosh(X) ** -1 * (1 - 0.5 * np.cos(2 * T)) +
     (np.cosh(X) ** -1 * np.tanh(X)) * (1 - 0.5 * np.sin(2 * T))
```

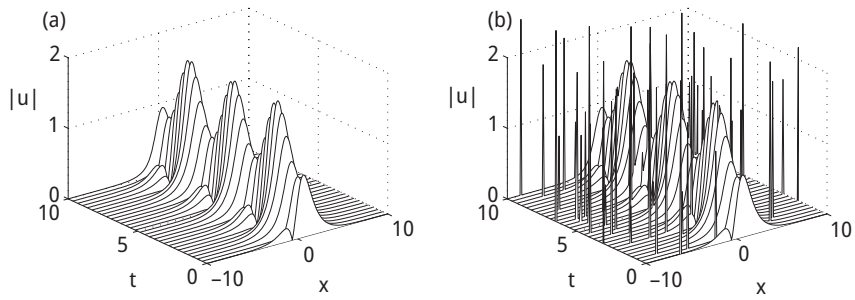


Figure 15.11 Representation of the space–time dynamics considered in Figs. 15.9 and 15.10. In (a), the ideal dynamics is demonstrated, while in (b), a corrupted (noisy) image is assumed to be generated by the data collection process, for instance. The addition of noise brings into question the ability of the SVD to produce meaningful results. For this example, the low-rank matrix $\mathbf{L}$ has two modes of significance, while the sparse matrix $\mathbf{S}$ has 60 non-zero entries.

To add sparse noise, the **randintrlv** is used to produce noise spikes (60 spikes in total) in certain matrix/pixel locations, thus corrupting the data matrix.

```
sam = 60
Atest2 = np.zeros((len(t), n))
Arand1 = np.random.rand(len(t), n)
Arand2 = np.random.rand(len(t), n)
r1 = np.random.permutation(len(t) * n)
r1k = r1[:sam]
for j in range(sam):
    Atest2[r1k[j] // n, r1k[j] % n] = -1

Anoise = Atest2 * (Arand1 + 1j * Arand2)
unoise = usol + 2 * Anoise
```

Figure 15.11 shows the ideal data along with the corrupt, or more physically relevant, data that might be collected in practice. The complex white noise fluctuations are, of course, the problematic part of the dimensionality reduction issue. Recall that without such noise fluctuations, the ideal data can be faithfully approximated with a low-rank, two-mode approximation.

The PCA reduction of the data matrices can now be compared by using the SVD. The following code produces the SVD decomposition of both data matrices:

```
A1=usol.T
A2=unoise.T
U1, S1, V1 = np.linalg.svd(A1)
U2, S2, V2 = np.linalg.svd(A2)
```

Both the singular values and the modal structures are represented in Figs. 15.12 and 15.13, respectively. Specifically, the top panel of Fig. 15.12 shows the two-mode dominance (top panel of Fig. 15.13) of the ideal data, while the middle panels of these figures show the singular value distribution and the first four modes, respectively. Note that the addition of the corrupt data activates many more modes. In fact, it takes 26 modes to capture 99% of the energy of the data compared to two modes previously. The modes are also highly perturbed from their ideal states due to the sparse (corrupt) noise that was added.

To apply the robust PCA algorithm, the corrupt data matrix is decomposed into real and imaginary parts before applying the **inexact_alm_rpca** algorithm. This algorithm effectively separates the data into low-rank and sparse components as given by (15.6.1). The low-rank portion is kept in order to then acquire the PCA modes necessary for reconstruction.

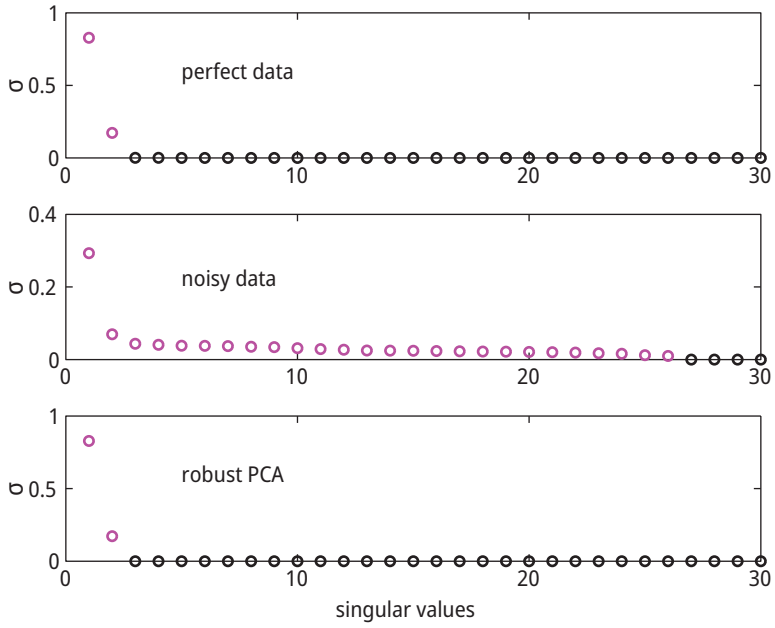


Figure 15.12 Singular values of the data matrices for the ideal data (top panel), the corrupted data (middle panel), and the low-rank data computed through robust PCA (bottom panel). The robust PCA construction produces almost a perfect match to the original, ideal data. The magenta dots represent the number of modes necessary to construct 99% of the original data. In this case, the ideal and robust PCA require two modes, while the corrupt data requires 26 modes.

```
ur = np.real(unoise)
ui = np.imag(unoise)

lambda_val = 0.2
R1r, R2r = rpca.ialm.fit(ur.T, lambda_val)
R1i, R2i = rpca.ialm.fit(ui.T, lambda_val)

R1 = R1r + 1j * R1i
R2 = R2r + 1j * R2i

U3, S3, V3 = svd(R1.conj().T)
```

Note here that the real and imaginary portions are put back together at the end of the algorithm in order to perform SVD of the low-rank portion. Further, the two matrices corresponding to the low-rank ($R1$) and sparse ($R2$) portions of the data matrix are plotted. The parameter $lambda$ used in the **inexact_alm_rpca** algorithm can be tuned to best separate

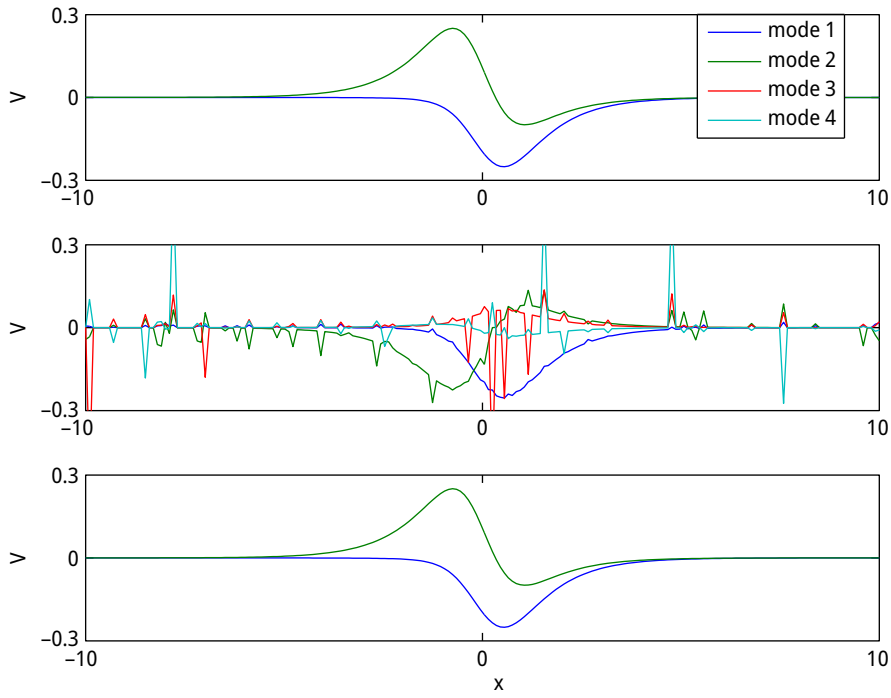


Figure 15.13 Dominant modes for the ideal data (top panel), the corrupted data (middle panel), and the low-rank data computed through robust PCA (bottom panel). The ideal and robust PCA produce the same dominant two modes, while the corrupt data produces highly erratic modes.

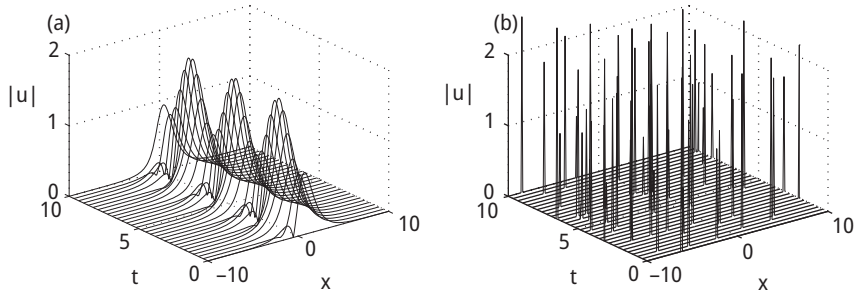


Figure 15.14 Separation of the original corrupt data matrix shown in Fig. 15.11(b) into a low-rank component (a) and a sparse component (b). The separation is almost perfect, with the low-rank matrix being dominated by two modes, i.e. they contain greater than 99% of the energy.

the sparse from low-rank matrices in (15.6.1). Here a value of 0.2 is used, which produces an almost ideal separation as is shown in Fig. 15.14. Indeed, an almost perfect separation is achieved as advertised (guaranteed) by Candés and co-workers [68].

To further investigate the robust PCA algorithm and its ability to extract the meaningful, low-rank matrix from the full matrix corrupted by sparse fluctuations, a series of low-rank

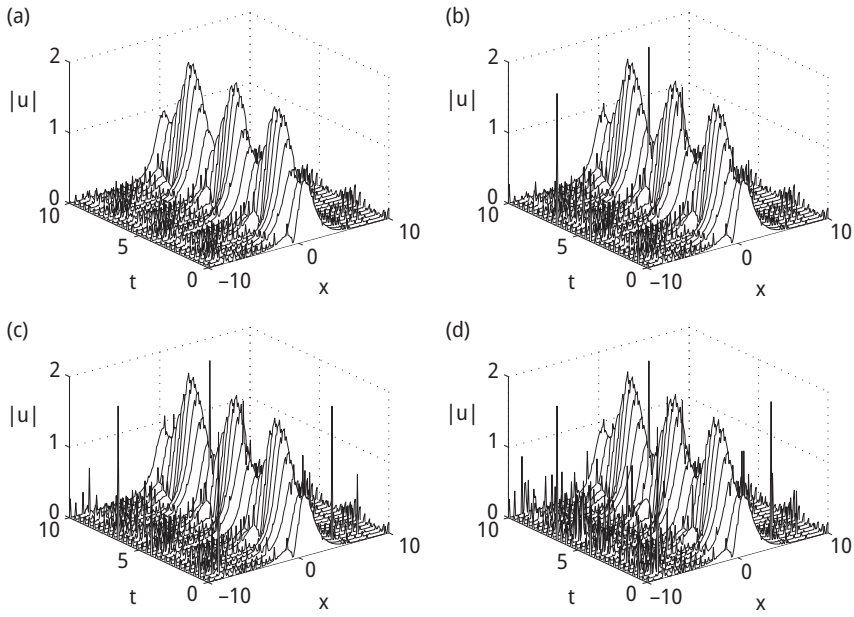


Figure 15.15 Two-, three-, four-, and five-mode reconstruction of the matrix using the corrupted (sparse plus low-rank) data. Note that the addition of the higher modes adds sparse data spikes as is expected since they have not been filtered out of the data matrix.

approximations is made of the data matrices using the PCA modes generated by the SVD (see Figs. 15.15 and 15.16). The low-rank approximations show how effective the robust PCA algorithm is in separating out the low-rank from sparse components.

Finally, to end this discussion, the effects of the parameter $lambda$ in the **inexact_alm_rpca** algorithm are illustrated. This parameter takes on values of zero to unity and can be tuned to achieve better or worse performance. The default, if $lambda$ is not included, does a reasonable job generically in separating the low-rank from the sparse matrices. As $lambda$ goes to zero, the computed low-rank matrix picks up the entire initial data matrix, i.e. the resulting sparse matrix is computed to be zero while the resulting sparse matrix contains the original data matrix. As $lambda$ goes to unity, the opposite happens with the computed low-rank matrix containing nothing while the sparse matrix contains virtually the entire original matrix. Figure 15.17 depicts the results of the **inexact_alm_rpca** algorithm for $lambda$ equal to 0.5 and 0.8. Note that as $lambda$ is increased, the fidelity of the low-rank matrix is compromised and corrupted by the sparsity.

As a final comment, the robust PCA algorithm advocated for here is quite remarkable in its ability to separate sparse corruption from low-rank structure in a given data matrix. The application of robust PCA essentially acts as an advanced filter for cleaning up a data matrix. Indeed, one can potentially use this as a filter which is very unlike the time–frequency filtering schemes considered previously.

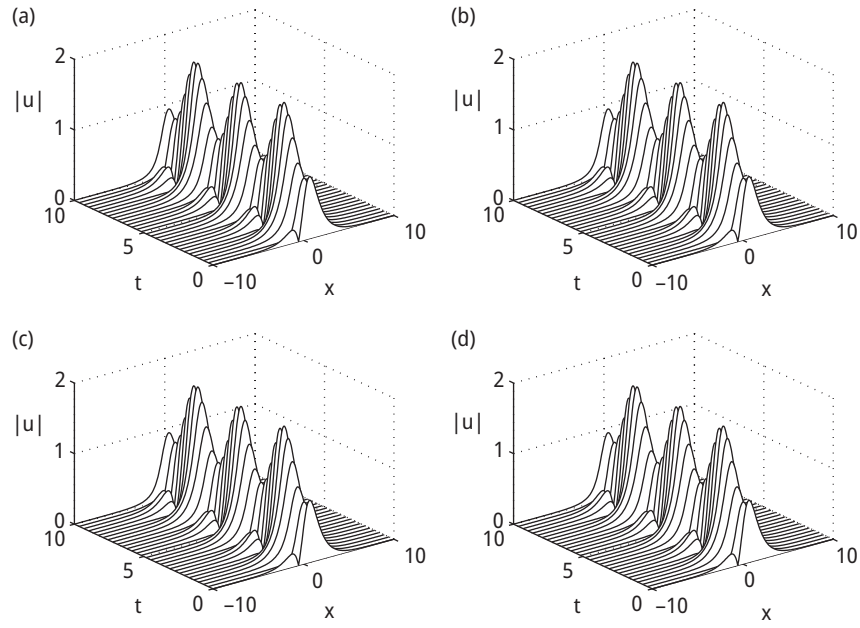


Figure 15.16 Two-, three-, four-, and five-mode reconstruction of the matrix using the robust PCA algorithm for extracting the corrupted (sparse) components. Note that a two-mode expansion is sufficient to capture all the features of interest.

15.7 Dynamic Mode Decomposition (DMD)

Dynamic Mode Decomposition (DMD) [73] is a data-driven algorithm that requires no underlying governing equations, rather snapshots of experimental or computational measurements are used to predict and control a given system [74–77]. In atmospheric sciences, a version of the DMD method is used to fit a set of a data to an underlying (best statistical fit) linear stochastic model. This is known as *linear inverse models* (LIMs) [78–80]. Thus, the methodology presented here uses data alone as the source for informing us of the state and dimensionality of the system.

The DMD method provides a decomposition of experimental data into a set of dynamic modes that are derived from snapshots of the data in time. The mathematics underlying the extraction of dynamic information from time-resolved snapshots is closely related to the idea of the Arnoldi algorithm, one of the workhorses of fast computational solvers. To set the stage, we begin with the data collection process. Two important parameters are required:

$$N = \text{number of spatial points saved per time snapshot} \tag{15.7.1}$$

$$M = \text{number of snapshots taken} \tag{15.7.2}$$

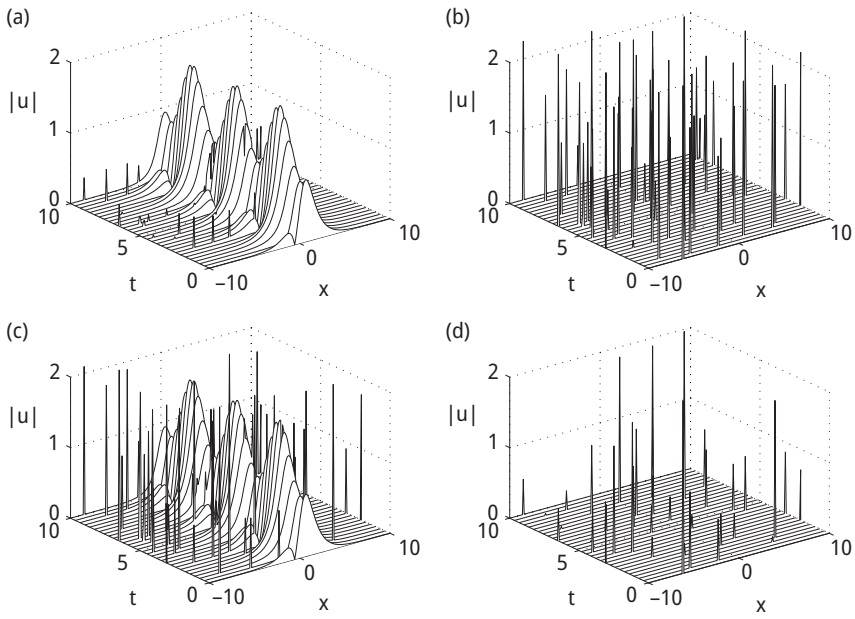


Figure 15.17 Separation of the original corrupt data matrix shown in Fig. 15.11(b) into a low-rank component (a) and (c) and a sparse component (b) and (d). The top panels are for $lambda = 0.5$ in the **inexact_alm_rpca** algorithm, while the bottom panels are for $lambda = 0.8$. The separation deteriorates from the almost perfect separation illustrated previously with $lambda = 0.2$.

The initial DMD algorithms required regularly spaced intervals of time. However, modern variants of DMD, specifically optimized DMD [82; 83], simply specify the time measurement locations

$$\text{data collection times}: \quad t_1, t_2, \cdots, t_M. \tag{15.7.3}$$

If the data is evenly spaced so that $t_{k+1} = t_k + \Delta t$, then the original version of DMD proposed by Schmid can be used [74; 75]. The data can then be arranged into an $N \times M$ matrix

$$\mathbf{X} = \begin{bmatrix} | & | & | & | & | \\ U(\mathbf{x}, t_1) & U(\mathbf{x}, t_2) & U(\mathbf{x}, t_3) & \cdots & U(\mathbf{x}, t_M) \\ | & | & | & | & | \end{bmatrix} \tag{15.7.4}$$

where $\mathbf{x}$ is a vector of data collection points of length N.

There are numerous variants of the DMD algorithm. However, we highlight here the optimized DMD algorithm, which is the most stable, unbiased, and robust regression for fitting the data matrix $\mathbf{X}$ by exponentials [82; 83]

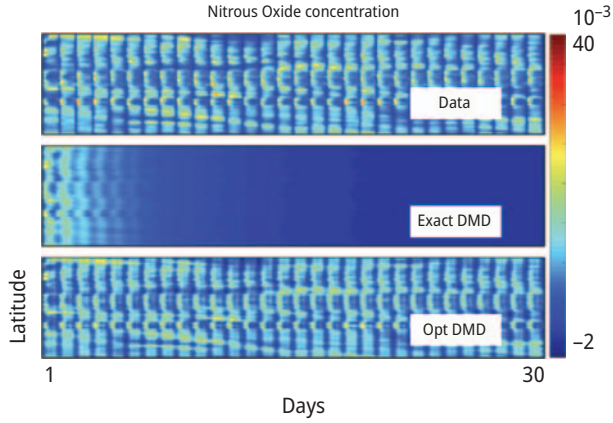


Figure 15.18 Canonical bias effect in DMD algorithms for noisy (normalized) data (top panel) for a specific longitude and elevation. In this example of atmospheric chemistry from Velagar et al. [81], 30 days of nitrous oxide (NO) are measured and a DMD model regression is used to fit the data, which has been normalized [81]. The bias of the exact DMD algorithm (middle panel) shows that the solution almost immediately tends to zero, while optimized DMD (bottom panel) is able to correctly approximate the chemical dynamics.

$$\text{argmin}_{\boldsymbol{\omega},\boldsymbol{\Phi},\mathbf{b}}\|\mathbf{X}-\sum_{j=1}^{\mathrm{r}}\mathrm{b_j}\boldsymbol{\Phi}_\mathrm{j}\mathrm{e}^{\omega_\mathrm{j}\mathrm{t}}\| = \text{argmin}_{\boldsymbol{\omega},\boldsymbol{\Phi},\mathbf{b}}\|\mathbf{X}-\boldsymbol{\Phi}\exp(\boldsymbol{\Omega}\mathrm{t})\boldsymbol{b}\|, \tag{15.7.5}$$

where an r-rank exponential approximation to a collection of state-space measurements $\mathbf{x}_k = \mathbf{x}(t_k)\ (k = 1, 2, \cdots, n)$ is performed. The algorithm regresses to values of the DMD eigenvalues ω_j, DMD modes $\boldsymbol{\Phi}_j$, and their respective loadings b_j. The ω_j determines the temporal behaviour of the system associated with a modal structure $\boldsymbol{\Phi}_j$, thus giving a highly interpretable representation of the dynamics. DMD may be thought of as a combination of SVD/POD in space with the Fourier transform in time, combining the strengths of each approach [77; 84]. Importantly, bias induced by measurement noise has significantly limited the forecasting and reconstruction performance of DMD algorithms, as illustrated in the example measurements of atmospheric chemistry dynamics of Fig. 15.18. As shown in Fig. 15.19, additionally using an optimized DMD formulation, the statistical method of bagging (**b**ootstrap **agg**regat**ing**) [85] significantly improves DMD robustness and accuracy for forecasting while also providing *uncertainty quantification* (UQ) in its predictions and reconstructions [83]. DMD is modular due to its simple formulation in terms of linear algebra, resulting in innovations related to control [86], compression [87; 88], reduced-order modelling [89; 90], and multi-resolution analysis [91; 92], among others.

The advantage of using DMD over SVD is that the DMD modes are linear combinations of the SVD modes that have a common linear (exponential) behaviour in time, given by oscillations at a fixed frequency with growth or decay. Specifically, given the optimization (15.7.5), DMD gives a simple model for the dynamics as

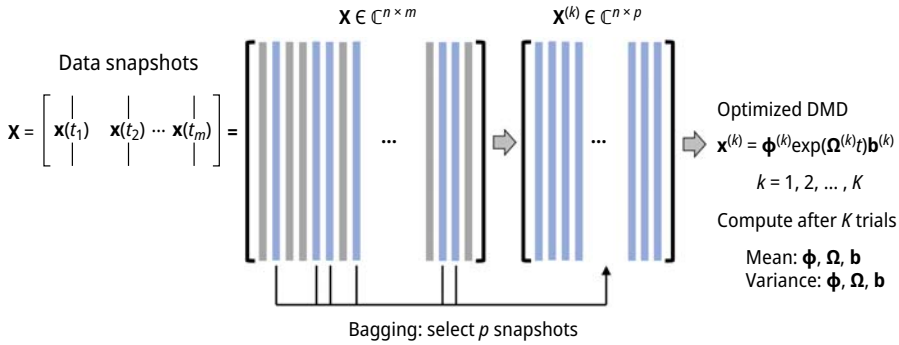


Figure 15.19 Summary of the BOP–DMD architecture. The data snapshots $\mathbf{x}(t_k)$ are collected over m snapshots into the matrix $\mathbf{X}$. Columns of $\mathbf{X}$ are randomly sub-selected into the matrix $\mathbf{X}^{(k)}$ to build an optimized DMD model. Each DMD model $\mathbf{x}^{(k)} = \mathbf{\Phi}^{(k)} \exp(\mathbf{\Omega}^{(k)} t)\mathbf{b}^{(k)}$ is used to compute the statistics (mean and variance) of the DMD parametrizations $\mathbf{\Phi}, \mathbf{\Omega}, \mathbf{b}$, which are used in building the BOP–DMD ensemble solution with UQ.

$$\mathbf{x}(t) = \sum_{j=1}^{r} \mathbf{\Phi}_j e^{\omega_j t} b_j = \mathbf{\Phi} \exp(\mathbf{\Omega} t)\mathbf{b}. \tag{15.7.6}$$

In addition, the BOP–DMD gives UQ metrics by producing the statistics of the modes, specifically the mean and variance, when bagging:

$$\text{mean, variance: } \langle \mathbf{\Phi} \rangle, \langle \mathbf{\Phi}^2 \rangle \tag{15.7.7}$$

$$\text{mean, variance: } \langle \mathbf{\Omega} \rangle, \langle \mathbf{\Omega}^2 \rangle \tag{15.7.8}$$

$$\text{mean, variance: } \langle \mathbf{b} \rangle, \langle \mathbf{b}^2 \rangle. \tag{15.7.9}$$

Thus, forecasting and reconstruction can be produced by drawing from the above distributions, making Monte Carlo forecasting, for instance, quite easy to achieve.

15.7.1 pyDMD package

The pyDMD package [93] provides a flexible, robust, and well-maintained Python package for executing the DMD algorithm. Its recent upgrade also includes the optimized DMD algorithm, which is critically enabling for use with real data [82; 83]. The algorithm can be easily installed with **pip install pydmd**. To execute a DMD fit on a data matrix, the following code is used. Importantly, two features are now available. First is the ability to do a time-delay embedding in order to correctly extract standing wave features in the data [94]. Second is the ability within the variable projection algorithm of optimized DMD to constrain the eigenvalues.

```
from pydmd import DMD, BOPDMD
from pydmd.plotter import plot_eigs, plot_summary
from pydmd.preprocessing import hankel_preprocessing

delay_optdmd = hankel_preprocessing(BOPDMD(svd_rank=20, num_trials=0,
     eig_constraints={"imag", "conjugate_pairs"}), d=2)
delay_optdmd.fit(U, t=t[1:])
plot_summary(delay_optdmd, index_modes=[0,2,4], order='F')
```

In the above example, a single time delay is used with constraints on the eigenvalues of purely imaginary eigenvalues that come in complex conjugate pairs. Everything is also plotted in a summary graph shown in Fig. 15.20. The constraints are quite useful in practice and the following combinations are possible.

```
eig_constraints={"imag", "conjugate_pairs"}
eig_constraints={"stable", "conjugate_pairs"}
eig_constraints={"conjugate_pairs"}
eig_constraints={"imag"}
eig_constraints={"stable"}
```

No constraint can also be specified by simply removing the line from the code. If no constraint is specified, or if only conjugate pairs are specified, then it is often the case with noisy data that eigenvalues can have real positive parts which will generate exponentially growing solutions in a forecast. For periodic, or quasi-periodic data, it is often practically useful to begin with the first constraints above of imaginary eigenvalues coming in complex conjugate pairs. Unless a multiscale [91, 92, 95] or nonstationary [96] architecture is used, DMD is not good for capturing transients.

To generate a forecast, pyDMD offers a forecasting call that needs to only specify the time vector desired for a forecast. The forecasting tool can also just as easily be used for reconstruction by putting in the original time vector used in the regression.

```
forecast_mean = delay_optdmd.forecast(time_forecast)
```

One can also easily access the various components of the DMD, including the critical components $\mathbf{\Phi}, \mathbf{\Omega}$, and $\mathbf{b}$, which are used in building a the BOP–DMD ensemble solution with UQ.

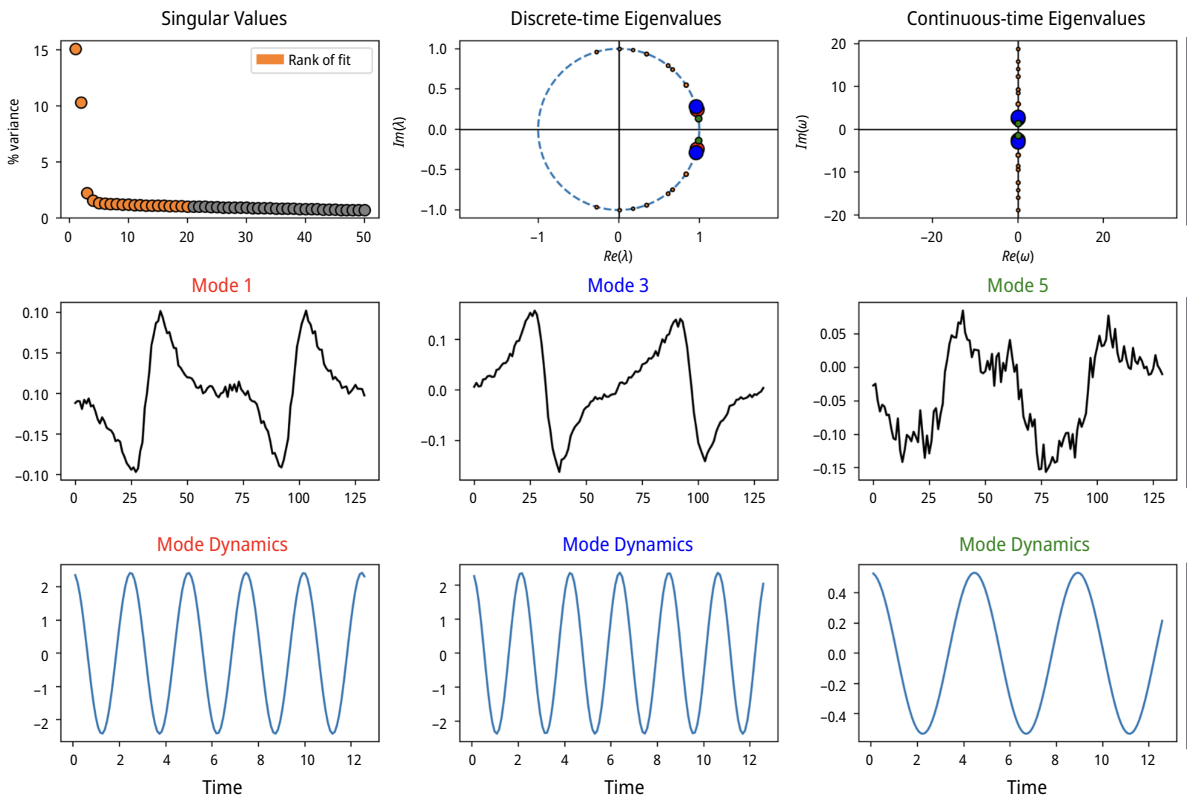


Figure 15.20 The plot summary of the DMD algorithm from the pyDMD package. The top left shows the singular value spectrum decay of the data and gives an indication of how many modes should be selected in the DMD algorithm. The top middle and top right panels show the continuous and discrete spectra of the DMD algorithm. The top right is the standard view of the algorithm as with the exponential representation as given by (15.7.6). The middle rows are the selected modes for viewing. In this case, complex conjugate pairs have enforced and what is visualized is the first three modes. However, because of the single time delay used, the mode is repeated. The bottom row shows the temporal behaviour associated with each mode plotted above it.

```
bn = delay_optdmd.amplitudes # Order the results by bn value
order = np.argsort(bn)[::-1] # Create the index of order
bn = bn[order] # Reorder bn

all_Psi = delay_optdmd.modes # Reorder Psi
Psi = all_Psi[:nx,:] # Only take the first nx (with delay embed of 1)
Psi = Psi[:,order]

Lambda = delay_optdmd.eigs  # Reorder Lambda
Lambda = Lambda[order]
```

Figure 15.20 shows the summary for the components $\mathbf{\Phi}, \mathbf{\Omega}$, and $\mathbf{b}$. It also allows for flexible visualization and exploration of the dominant mode structure in the dynamics.

For a specific example, a spatio-temporal field is created and noise added. The field will contain two modes with two distinct oscillatory frequencies

$$u(x,t) = \text{sech}(x)\cos(2.3t) + \text{sech}(x)\tanh(x)\sin(2.8t). \tag{15.7.10}$$

Although there are two modes, this is actually four modes in the DMD representation, as the cosine and sine can be represented in the complex notation (15.7.6). Specifically, the field is given by

$$u(x,t) = \frac{1}{2}\text{sech}(x)\left[\exp(i2.3t) + i\exp(-i2.3t)\right] + \frac{1}{i}\text{sech}(x)\tanh(x) \left[\exp(i2.8t) - i\exp(-i2.8t)\right]. \tag{15.7.11}$$

So in a DMD decomposition, at least four modes are required to get these modes constructed correctly. This is done with the **svd_rank** setting in the optimized DMD algorithm. Noise is added to the field in order to make the task of reconstruction and forecasting more difficult. Indeed, noise is what compromises the variants of the DMD algorithm. Figure 15.21 shows the spatio-temporal field, its reconstruction, and forecasting capabilities.

15.8 Koopman Operators

Koopman theory is directly related to DMD, having been shown to be a numerical approximation to the Koopman operator [76]. This connection has been critical in the overall development of Koopman operator theory and its various numerical approximations. Koopman theory [97] more broadly has focused on finding a better representation, or coordinate

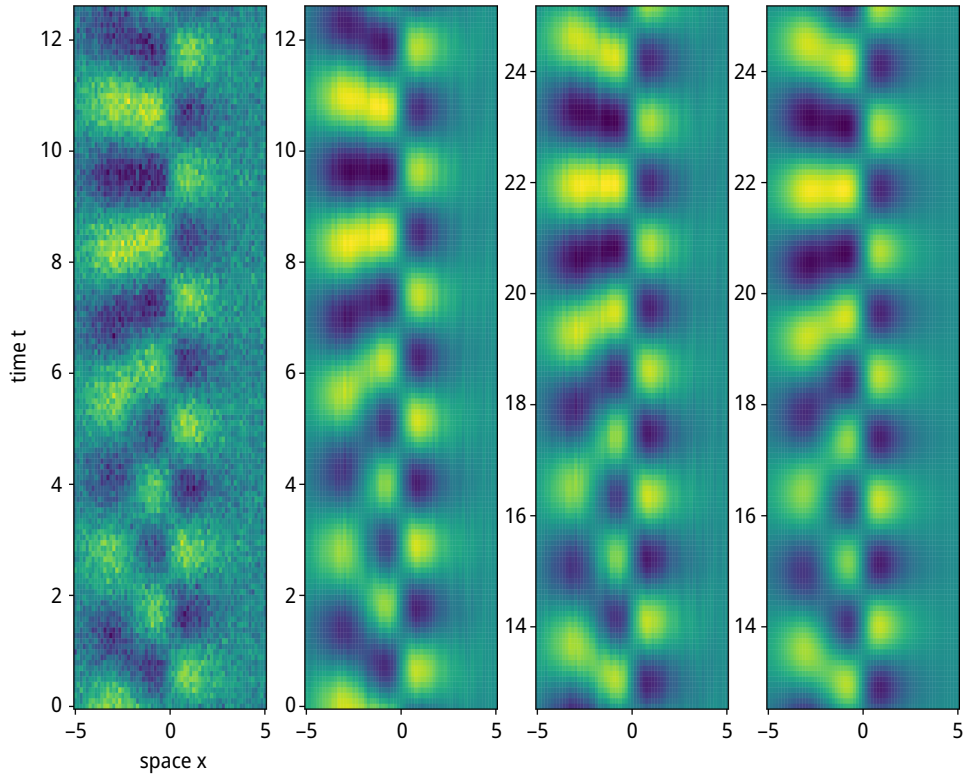


Figure 15.21 Optimized DMD reconstruction of noisy spatial temporal data (left panel). The second panel is the optimized DMD reconstruction, while the third panel is the optimized DMD forecast, which is compared with the ground truth in the far right panel. The optimized DMD is able to overcome the noise-induced bias, which often drives the solution to zero as shown in Fig. 15.21.

system, for which the exponential regression provides an improved linear model. So instead of working with $\mathbf{x}$ directly, we can instead find coordinates $\mathbf{z} = \boldsymbol{\varphi}(\mathbf{x})$ such that

$$\frac{d}{dt}\mathbf{z} = \mathbf{A}\mathbf{z} \tag{15.8.1}$$

is a much better approximation to the true dynamics than simply applying DMD to $\mathbf{x}$. The transformation $\boldsymbol{\varphi}(\cdot)$ is thus an approximation to a linearizing transformation. Such linearizing transformations are rare to discover analytically, but famous examples include the Cole–Hopf transformation for linearizing viscous Burgers into the heat equation [98; 99]. Approximately two decades later, the *inverse scattering transform* (IST) was developed to linearize integrable PDEs such as the Korteweg deVries and nonlinear Schrödinger equation

[100]. It has been more than five decades since principled analytic methods have emerged for linearizing ODEs and PDEs.

Data-driven modelling has centred instead on finding Koopman operators directly from measurement data. The data collected can be equivalently cast as learning a flow map $\mathbf{x}_{k+1} = \mathbf{F}(\mathbf{x}_k)$, where $\mathbf{x}_k = \mathbf{x}(t_k) = \mathbf{x}(k\Delta t)$. The goal is then to find a linearizing coordinate transform so that $\mathbf{z}_{k+1} = \mathbf{K}\mathbf{z}_k$ is an improved model for reconstruction and forecasting. These coordinates are given by eigenfunctions of the discrete-time Koopman operator, which advances a measurement function $g(\mathbf{x})$ of the state forward in time through the dynamics:

$$\mathcal{K}g(\mathbf{x}_k) = g(\mathbf{F}(\mathbf{x}_k)) = g(\mathbf{x}_{k+1}). \tag{15.8.2}$$

For an eigenfunction φ of $\mathcal{K}$, corresponding to an eigenvalue λ, this becomes

$$\mathcal{K}\varphi(\mathbf{x}_k) = \lambda\varphi(\mathbf{x}_k) = \varphi(\mathbf{x}_{k+1}). \tag{15.8.3}$$

Thus, approximating the spectral decomposition from measurement data is again the focal point, which is what the DMD algorithm is geared for. Thus, DMD is then applied in the new latent variable. Alternatively, neural networks, [101–103], diffusion maps [104], and time-delay embeddings [105–107] all allow for the data-driven construction of mappings capable of transforming nonlinear PDEs into linear PDEs whose Koopman operator can be constructed.

We can use the optimized DMD method of the previous section as the basic underlying algorithm in the Koopman operator construction. Although there are significant theoretical considerations in Koopman theory about projecting into infinite-dimensional operator space [97; 108], the aim here will be to consider practical numerical approximations of Koopman that are finite-dimensional. So in considering (15.8.2), two methods will be shown to lift to higher-dimensional, but finite, spaces. First, we construct a new latent variable

$$\mathbf{z}_k = \begin{bmatrix} \mathbf{x}_k \\ g_1(\mathbf{x}_k) \\ g_2(\mathbf{x}_k) \\ \vdots \\ g_p(\mathbf{x}_k) \end{bmatrix} \tag{15.8.4}$$

where there are p total new *observable* vectors $g_j(\cdot) \in \mathbb{R}^n$. Snapshots of the latent dynamics are

$$\mathbf{Z} = \begin{bmatrix} | & | & | & | & | \\ \mathbf{z}_1 & \mathbf{z}_2 & \mathbf{z}_3 & \cdots & \mathbf{z}_M \\ | & | & | & | & | \end{bmatrix} \tag{15.8.5}$$

DMD gives a simple model for the latent dynamics as

$$\mathbf{z}(t) = \sum_{j=1}^{r} \mathbf{\Phi}_j e^{\omega_j t} b_j = \mathbf{\Phi} \exp(\mathbf{\Omega} t)\mathbf{b}. \tag{15.8.6}$$

In addition, recall that the BOP–DMD gives UQ metrics by producing the statistics of the modes, specifically the mean and variance. Currently, there are no principled ways of choosing the observables $g_j(\cdot)$. Extended and kernel DMD [109; 110] give options to use ideas from support vector machines and kernel methods. These models must be carefully trained in order to avoid producing a stable model. Unstable models are common and result directly from the fact that many observables $g_j(\cdot)$ give rise to unstable growth modes.

Time-delay embeddings provide a highly effective alternative to enriching the observables [105; 106]. More than that, there is a direct connection to Takens' embedding theorem [111] and a proof that for long time-delay embeddings, the model converges to a discrete Fourier transform in time which is a linear model [112]. Time-delay embeddings thus trade out the matrix (15.8.4) for the matrix

$$\mathbf{z}_k = \begin{bmatrix} \mathbf{x}_k \\ \mathbf{x}_{k-1} \\ \mathbf{x}_{k-2} \\ \vdots \\ \mathbf{x}_{k-p} \end{bmatrix} \tag{15.8.7}$$

which then gives the Hankel matrix

$$\mathbf{Z} = \mathbf{H} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{M-q} \\ \mathbf{x}_2 & \mathbf{x}_3 & \mathbf{x}_4 & \cdots & \mathbf{x}_{M-q+1} \\ | & | & | & | & | \\ \mathbf{x}_q & \mathbf{x}_{q+1} & \mathbf{x}_{q+2} & \cdots & \mathbf{x}_M \end{bmatrix}. \tag{15.8.8}$$

We simply apply optimized DMD to either (15.8.5) or (15.8.8).

An important note, in (15.8.4) and also in (15.8.7), the first n-dimensions of the observable is the original observable $\mathbf{x}$. This is critical, as upon completion of the DMD algorithm, mapping back to the original variables is important [113]. So once a Koopman approximation (15.8.6) is produced, only the first n components of $\mathbf{\Phi}$ are required for approximating the DMD mode in the original coordinates.

To demonstrate two simple Koopman representations, the nonlinear Schrödinger (NLS) equation is considered

$$iu_t + \frac{1}{2}u_{xx} + |u|^2 u = 0 \tag{15.8.9}$$

with the initial 2-soliton solution $u(x,0) = \text{sech}(x)$. This produces a strongly nonlinear spatio-temporal evolution, which we can approximate with a DMD and Koopman representation. The evolution dynamics can be generated as follows.

```
m = 250; n = 256; l = 20.0
t = np.linspace(0,6,2*m); dt = t[1]-t[0]
x = np.linspace(-l/2,l/2,n+1)[0:n]; dx = x[2]-x[1]

def nls_rhs(ut_sep, t, params):
    n, k = params
    ut = ut_sep[0:n] + 1j*ut_sep[n:]
    u = ifft(ut)
    deriv = -1j/2.0*(k**2)*ut + 1j*fft((abs(u)**2)*u)
    return np.hstack([np.real(deriv),np.imag(deriv)])

k = 2*np.pi*np.fft.fftfreq(n, d = float(l)/n)
params = (n, k)

u0 = 2*np.cosh(x)**-1
ut0= np.hstack([np.real(fft(u0)),np.imag(fft(u0))])
ut = odeint(nls_rhs, ut0, t, args=(params,), rtol=1e-10, atol=1e-10)
ut = (ut[:,0:n] + 1j*ut[:,n:]).T

u = np.zeros((n,2*m), dtype = complex)
for j in range(2*m):
    u[:,j]=ifft(ut[:,j]);

random_matrix = np.random.normal(0, 0.2, size=(n, 2*m))
u = (u + random_matrix).T
```

Noise is added to the data to make it more challenging for the DMD/Koopman algorithm. The training window for the algorithm will be the first half of the time evolution $t \in [0,3]$. The algorithm can then be tested for both reconstruction and forecasting.

For the first Koopman representation, the following observables are used:

$$\mathbf{z}_k = \begin{bmatrix} \mathbf{x}_k \\ g_1(\mathbf{x}_k) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_k \\ |\mathbf{x}_k|^2 \mathbf{x}_k \end{bmatrix}. \tag{15.8.10}$$

The choice of observables is motivated by the NLS form of nonlinearity. The Koopman operator is then trivially constructed with the pyDMD package.

```
u1 = u[:m,:].T
u3 = ( ( np.abs(u1)**2 )*u1 )
U  = np.vstack((u1, u3))

delay_optdmd = hankel_preprocessing(BOPDMD(svd_rank=5, num_trials=0,
     eig_constraints={"imag", "conjugate_pairs"}), d=2)
delay_optdmd.fit(U, t=t[1:m])
```

The Koopman operator for reconstruction and forecasting can be evaluated from the following code block:

```
time_recon = t[:m]
time_forecast = t[m:]

recon_mean = delay_optdmd.forecast(time_recon)
forecast_mean = delay_optdmd.forecast(time_forecast)

Recon = recon_mean[:n,:].T
Fore = forecast_mean[:n,:].T
```

Alternatively, time-delay embedding can be used to construct an approximate Koopman operator. This can be done by simply using more delays in the pyDMD package. The following code segment shows how to build a Koopman approximation using 10 time-delay embeddings.

```
U = u[:m,:].T
delay_optdmd = hankel_preprocessing(BOPDMD(svd_rank=5, num_trials=0,
      eig_constraints={"imag", "conjugate_pairs"}), d=10)
delay_optdmd.fit(U, t=t[9:m])
```

Once computed, only the first n-components of the model need to be extracted for estimating the original spatio-temporal variable. Although it is fairly easy to implement either an augmented observable space or create a time-delay embedding for the Koopman operator, the time-delay embedding is recommended, as it is generally a more robust method for producing a good Koopman approximation. In contrast, the choice of observables $g_k(\cdot)$ can lead to difficulties in producing a good Koopman approximation. This remains a difficult

task, as there are currently no good and/or principled methods for selecting observables. Moreover, selecting a large number of observables can make the data matrix exceptionally large in practice.

15.9 Randomized Linear Algebra and Scalable SVD and DMD

The SVD and DMD algorithms provide exceptional frameworks for finding low-rank structure and features in data matrices. Moreover, reconstruction of the data is accomplished using the powerful concept of *linear superposition* of the SVD/DMD modes. For a square data matrix $\mathbf{A}$ of size $N \times N$, the computational cost of the decomposition scales approximately as $O(N^3)$. For very large matrices, for instance where $N \sim 10^6$, this would result in a computation of size 10^{18}, which may be prohibitively large and/or intractable from a memory and time-requirement point of view. Yet in the modern data-driven era, it is not uncommon to work with data matrices that are significantly larger. Randomized and probabilistic linear algebra methods provide an exceptional advancement for extracting low-rank features from modern high-dimensional data sets.

Following the review and exposition of Erichson et al. [114], we again assume that a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has rank r, where $r \ll \min\{m, n\}$. Both SVD and DMD, in general, find a low-rank matrix approximation using smaller matrices. In the case of SVD, the low-rank subspace is spanned by the unitary matrix $\mathbf{U}$, while in DMD, the low-rank space is spanned by the DMD modes $\mathbf{\Phi}$. More generally, the generic structure of such decompositions is given by:

$$\underset{m \times n}{\mathbf{A}} \approx \underset{m \times r}{\mathbf{E}} \underset{r \times n}{\mathbf{F}}, \tag{15.9.1}$$

where the columns of the matrix $\mathbf{E} \in \mathbb{R}^{m \times r}$ span the column space of $\mathbf{A}$, and the rows of the matrix $\mathbf{F} \in \mathbb{R}^{r \times n}$ span the row space of $\mathbf{A}$. Thus, if the data matrix is spatio-temporal in nature, then $\mathbf{E}$ captures the spatial features of the data, while $\mathbf{F}$ captures the associated time dynamics. The factors $\mathbf{E}$ and $\mathbf{F}$, like in DMD and SVD, can then be used to summarize or to reveal some interesting structure in the data. Further, the $\mathbf{E}$ and $\mathbf{F}$ matrices can be used to efficiently store, or compress, the large data matrix $\mathbf{A}$. Specifically, while $\mathbf{A}$ requires $O(mn)$ of storage, $\mathbf{E}$ and $\mathbf{F}$ require only $O(mr + nr)$ of storage.

As with SVD and DMD, the rank truncation used in real data is a hyper-parameter that is tuned in order to balance reconstruction error with efficient representation. That is, we would like to choose the rank r as small as possible while retaining the dominant features of the data matrix. More precisely, we would like to find a rank-r matrix $\mathbf{A}_r$, which is as close as possible to an arbitrary input matrix $\mathbf{A}$ in the least-square sense. Thus, the r chosen in the process is known as the target rank. The probabilistic framework formulated by Halko et al. [115] allows one to compute a near-optimal low-rank approximation for a given target rank-r. This is done by splitting the computation into two logical stages as shown in Fig. 15.22:

- **Stage 1**: Construct a low-dimensional subspace that approximates the column space of $\mathbf{A}$. This means, the aim is to find a matrix $\mathbf{Q} \in \mathbb{R}^{m \times r}$ with orthonormal columns such that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top \mathbf{A}$ is satisfied.
- **Stage 2**: Construct a smaller matrix $\mathbf{B} := \mathbf{Q}^\top \mathbf{A} \in \mathbb{R}^{r \times n}$ by restricting the high-dimensional input matrix to the low-dimensional space spanned by the near-optimal basis $\mathbf{Q}$. The smaller matrix $\mathbf{B}$ is then used to compute a desired low-rank approximation.

The computation of $\mathbf{Q}$ is where randomness is leveraged. Once computed, the second stage is purely deterministic.

Stage 1 – Computing the near-optimal basis: The first step in the randomized algorithm is to approximate a near-optimal basis $\mathbf{Q}$ for the matrix $\mathbf{A}$ such that

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top \mathbf{A} \tag{15.9.2}$$

is satisfied. The desired target rank r is assumed to be $r \ll \min\{m, n\}$. Specifically, $\mathbf{P} := \mathbf{Q}\mathbf{Q}^\top$ is a linear orthogonal projector. A projection operator corresponds to a linear subspace, and transforms any vector to its orthogonal projection on the subspace.

Random projections can be used to sample the range (column space) of the input matrix $\mathbf{A}$ in order to efficiently construct such an orthogonal projector. Random projections are data agnostic, and constructed by first drawing a set of r random vectors $\{\boldsymbol{\omega}_i\}_{i=1}^r$, for instance, from the standard normal distribution. Probability theory guarantees that random vectors are linearly independent with high probability. These are used to generate a set of random projections $\{\mathbf{y}_i\}_{i=1}^r$, which map $\mathbf{A}$ to a low-dimensional space

$$\mathbf{y}_i := \mathbf{A}\boldsymbol{\omega}_i \quad \text{for } i = 1, 2, ..., r. \tag{15.9.3}$$

This process forms a set of independent randomly weighted linear combinations of the columns of $\mathbf{A}$, and reduces the number of columns from n to r. While the input matrix is compressed, the Euclidean distances between the original data points are approximately preserved. Random projections are also well grounded in theory and known as the Johnson–Lindenstrauss (JL) transform [116; 117].

Equation (15.9.3) can be efficiently executed in parallel, allowing it to scale efficiently for very large data matrices. We define the random test matrix $\boldsymbol{\Omega} \in \mathbb{R}^{n \times r}$, which is again drawn from the standard normal distribution, and the columns of which are given by the vectors $\{\boldsymbol{\omega}_i\}$. The sample matrix $\mathbf{Y} \in \mathbb{R}^{m \times r}$, also denoted as sketch, is then obtained by post-multiplying the input matrix by the random test matrix

$$\mathbf{Y} := \mathbf{A}\boldsymbol{\Omega}. \tag{15.9.4}$$

Once $\mathbf{Y}$ is obtained, it only remains to orthonormalize the columns in order to form a natural basis $\mathbf{Q} \in \mathbb{R}^{m \times k}$. This can be efficiently achieved using the QR-decomposition $\mathbf{Y} =: \mathbf{Q}\mathbf{R}$, and it follows that (15.9.2) is satisfied.

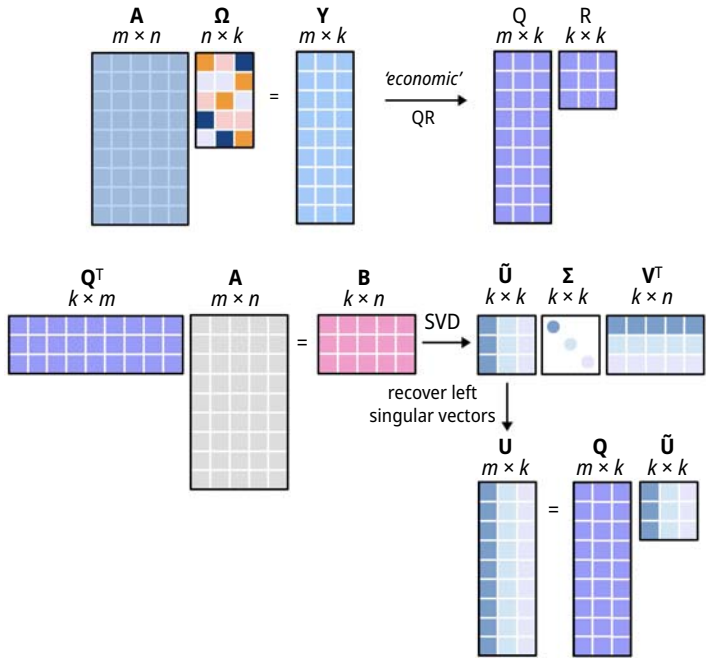


Figure 15.22 Conceptual architecture of the randomized singular value decomposition (rSVD). First, a natural basis $\mathbf{Q}$ is computed in order to derive the smaller matrix $\mathbf{B}$. Then, the SVD is efficiently computed using this smaller matrix. Finally, the left singular vectors $\mathbf{U}$ may be reconstructed from the approximate singular vectors $\tilde{\mathbf{U}}$ by the expression in (15.9.8). Reproduced with permission from Erichson et al. [114].

Stage 2 – Compute the smaller matrix Given the randomized computation of the near-optimal basis $\mathbf{Q}$, we aim to find a smaller matrix $\mathbf{B} \in \mathbb{R}^{r \times n}$ on which the data is projected. Specifically, we project the high-dimensional input matrix $\mathbf{A}$ to the near-optimal low-dimensional space

$$\mathbf{B} := \mathbf{Q}^\top \mathbf{A}. \tag{15.9.5}$$

Geometrically, this is a projection (i.e., a linear transformation), which takes points in a high-dimensional space into corresponding points in a low-dimensional space. Due to JL theory [116; 117], this preserves the geometric structure of the data in a Euclidean sense, i.e. the length of the projected vectors as well as the angles between the projected vectors are preserved. This is, due to the invariance of inner products [58]. Substituting (15.9.5) into (15.9.2) then yields the following low-rank approximation:

$$\underset{m \times n}{\mathbf{A}} \approx \underset{m \times k}{\mathbf{Q}} \underset{k \times n}{\mathbf{B}}. \tag{15.9.6}$$

This decomposition is referred to as the QB decomposition. Subsequently, the smaller matrix $\mathbf{B}$ can be used to compute a matrix decomposition using a traditional algorithm. More precisely,

$$\mathbf{B} = \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^* \tag{15.9.7}$$

where the SVD has been enacted on the massively reduced matrix $\mathbf{B}$. To get back to the original full state-space, only multiplication by $\mathbf{Q}$ is required. Specifically,

$$\mathbf{U} \approx \mathbf{Q}\tilde{\mathbf{U}} \tag{15.9.8}$$

approximates the original low-rank subspace.

The above two-stage reduction thus takes the construction of the SVD from the matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ to the matrix $\mathbf{B} \in \mathbb{R}^{r\times n}$, where $r \ll m$. In practice, two important innovations are usually applied as they greatly aid computational performance and accuracy. First, the algorithm allows for both oversampling ($l = r + p$), and additional power iterations q, in order to obtain the near-optimal basis matrix. Note that if an oversampling parameter $p > 0$ has been specified, the desired rank-r approximation is simply obtained by truncating the left and right singular vectors and the singular values. Default values for the oversampling and the power iteration scheme are often chosen to be $p = 10$ and $q = 2$, respectively.

A similar approach can be taken for the randomized DMD reduction [118; 119]. This allows for scaling to large data matrices, or even making SVD computations relatively fast on laptop-level computing. Randomized algorithms are part of the sklearn Python package. The following command executes a randomized SVD to compute the first 20 singular values and their associated time and space modes, i.e. the right and left eigenvectors, respectively:

```
from sklearn.utils.extmath import randomized_svd
u, s, v = randomized_svd(A, n_components=20, n_iter='auto')
```

The improvement in speed, and relatively little loss in accuracy, is quite remarkable [114]. Anecdotally, a matrix which takes 25–30 minutes on my laptop to compute an SVD takes about 20 seconds with the randomized algorithm. Thus, even more efficient interactive programming is achieved leveraging this method.

15.10 Autoencoders and Nonlinear SVD

The SVD and DMD algorithms are exceptional at providing a low-rank characterization of a data. They also have the advantage of being *linear* methods, which allows one to use linear superposition to construct solutions. Specifically, a low-rank reconstruction of the data

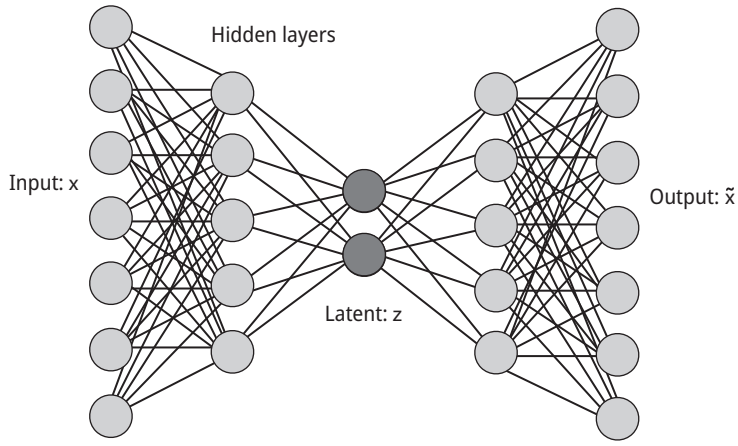


Figure 15.23 Basic architecture of an autoencoder. The input data is typically high-dimensional and the encoding passes through a specified number of layers into a latent space of r dimensions. This latent space is thus the nonlinear equivalent of an r-rank approximation of the data. The decoder projects the data back to the original high-dimensional space.

is achieved by a weighted sum of the r dominant modes. The hope is that the r-rank reconstruction is faithful to the data and thus allows for modelling a system with interpretable modal structures.

Autoencoders (AEs) [120–122] allow for a neural network generalization of the linear SVD decomposition. In concept, it is exactly the same. The neural network is constructed to learn an r-dimensional latent space that is capable of faithful reconstruction of the data. Figure 15.23 gives an example of an AE that moves from the original data space $\mathbf{x}$ to a new, r-dimensional latent space, and then decodes the data back to a reconstruction $\tilde{\mathbf{x}}$ of the original data. Thus, one of the critical aspects of the neural network is simply to preserve the reconstruction of the data through the *pinch* of the latent space

$$\text{argmin}_{\boldsymbol{\theta}} \|\mathbf{X} - \tilde{\mathbf{X}}\|_2^2 \tag{15.10.1}$$

where $\boldsymbol{\theta}$ are the weights of the neural network that are determined through optimization. To make connection back to the SVD, if the encoding was linear, then the latent space is given by the projection to the leading rSVD modes $\mathbf{z} = \mathbf{U}^*\mathbf{x}$, where $\mathbf{U}$ is a matrix with the first r columns of the SVD. To produce an approximation from the latent space to original data space, then $\tilde{\mathbf{x}} = \mathbf{U}\mathbf{z}$. In this interpretation, the latent space $\mathbf{z}$ is equivalent to the SVD representation of data in the low-rank space $\mathbf{\Sigma V}^*$. This is the space where clustering, classification, and regression is often performed with SVD/PCA, and this is exactly the same for the AE.

Autoencoders produce a nonlinear generalization of the SVD, generating a low-rank feature space via nonlinear projection to the latent space as shown in Fig. 15.23. To demonstrate the difference between SVD and AEs, consider the example data set of **yalefaces**. The data file has a total of 39 different faces with 65 lighting scenes for each face (2414 faces in all).

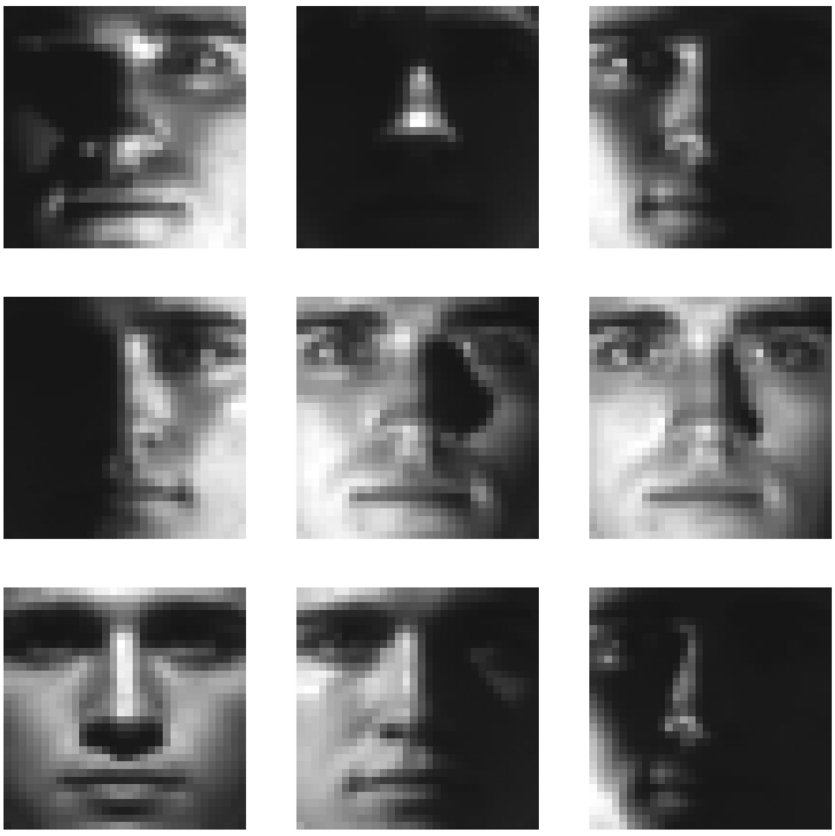


Figure 15.24 First nine faces in the **yalefaces** data set. The data file has a total of 39 different faces with 65 lighting scenes for each face (2414 faces in all). The individual images are columns of the matrix $\mathbf{X}$, where each image has been downsampled to 32×32 pixels and converted into greyscale with values between 0 and 1.

The individual images are columns of the matrix $\mathbf{X}$, where each image has been downsampled to 32×32 pixels and converted into greyscale with values between 0 and 1. The first nine faces in the data set are illustrated in Fig. 15.24. The total data matrix of interest is size 1024×2414. The data is saved in a compressed format and can be loaded and evaluated with the SVD as follows:

```
results=loadmat('yalefaces.mat')
X=results['X']
u,s,v=np.linalg.svd(X)
```

This produces the low-rank decomposition of the data. To see the results of this linear low-rank approximation, the first nine SVD modes are plotted in Fig. 15.25. These are the



Figure 15.25 Dominant correlated features of the aligned and cropped faces. These are the first nine reshaped columns of the matrix $\mathbf{U}$. The first mode (top left) shows the *average* face, or dominant correlated feature among all faces. The remaining modes produce features that are prevalent in the data. For instance, the second mode (middle top) is a lighting mode whereby one half of the face is illuminated. Figure 15.24 clearly shows that this is indeed a dominant feature.

dominant correlations among the 2414 faces in the data set. Importantly, the features are extracted by aligning the faces and cropping them in order for the SVD to construct a reasonable feature space. If the images were not aligned and cropped, this would lead to a translational and scaling invariance which is problematic for SVD-based methods. The extracted dominant modes are interpretable features of the data, highlighting the aspects of the data, including lighting conditions, which can be used for recognition tasks. Thus, the SVD was recognized early on as an exceptional feature extraction tool for facial recognition tasks [10, 11, 123].

To contrast the linear embedding of the images into a low-rank latent space, we construct an AE with three layers and ReLU activation functions. In this AE, the original state-space of 1024 dimensions is compressed in the encoder to 256, then 128, and then r dimensions. The decoder undoes this transformation by starting from r dimensions and then expanding to 128, then 256, and then finally 1024 dimensions.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

class Autoencoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Autoencoder, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, latent_dim),
            nn.ReLU()
        )
        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, input_dim),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

The AE hyper-parameters that can be tuned include the activation functions (selected to be ReLU here), the number of layers, and the size of each layer. Depending on the complexity of the task, the AE can be made quite sophisticated for the task at hand [102]. Here a basic AE is constructed without any enforcement or constraints on the latent space aside from the root-mean square reconstruction loss as given by (15.10.1).

Training the AE first requires putting the data in torch format, selecting the optimizer, the optimization loss, the batch size, and the learning rate. These are all hyper-parameters that can aid in the performance of the AE. The following code trains the AE.

```
X2 = (X.T).astype(np.float32)
tensor_X = torch.from_numpy(X2)

dataset = TensorDataset(tensor_X)
dataloader = DataLoader(dataset, batch_size=10, shuffle=True)

r = 20; [m,n] = X.shape; input_dim = m
autoencoder = Autoencoder(m, r)
optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)
loss_function = nn.MSELoss()

# Training loop
num_epochs = 500
for epoch in range(num_epochs):
    for data in dataloader:
        inputs, = data
        optimizer.zero_grad()
        outputs = autoencoder(inputs)
        loss = loss_function(outputs, inputs)
        loss.backward()
        optimizer.step()
    if epoch % 10 == 0:
        print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

Up to 500 epochs are selected for training. This will learn a latent representation of dimension $r = 50$. To see the features learned by the AE, we can look at what each latent node has encoded. Thus, to see what is encoded in the first node of the latent space of the AE, we simply put in the vector $\mathbf{z}_1 = [1, 0, 0, \cdots, 0]$ into the latent space and decode. The second node encoding can be found from $\mathbf{z}_2 = [0, 1, 0, \cdots, 0]$ and so-forth. Access to the feature space is then produced with the following code, which looks at the features encoded in the first nine latent variables.

```
latent_vectors = torch.eye(r)
latent_vectors2 = latent_vectors[:9] # pull 9 modes
decoded = autoencoder.decoder(latent_vectors2).detach().numpy()
```



Figure 15.26 Dominant correlated features of the aligned and cropped faces as learned by an autoencoder neural network. These are the first nine reshaped columns of the latent space $\mathbf{z}$. Unlike the SVD that allow for an easy understanding of the weightings for linear superposition in reconstruction, the decoder is a nonlinear function, which does not easily allow for understanding of the weightings of each of these features.

The plot of the first nine features are shown in Fig. 15.26. Unlike the SVD, the modes are not linearly superimposed. Thus, interpretation is significantly more difficult with the latent variables that encode the feature space of faces. Figure 15.27 shows the difference in approximation capabilities of the SVD versus the AE. Specifically, the AE is generally more accurate for a limited number of modes due to the flexibility and approximation capabilities of the neural network. The distribution of the data on the latent space is also shown with a box plot. Interestingly, the AE during training can zero out six nodes without any lack of performance. The simulations where done with only 100 epochs of training; thus, the AE results can be improved with further training.

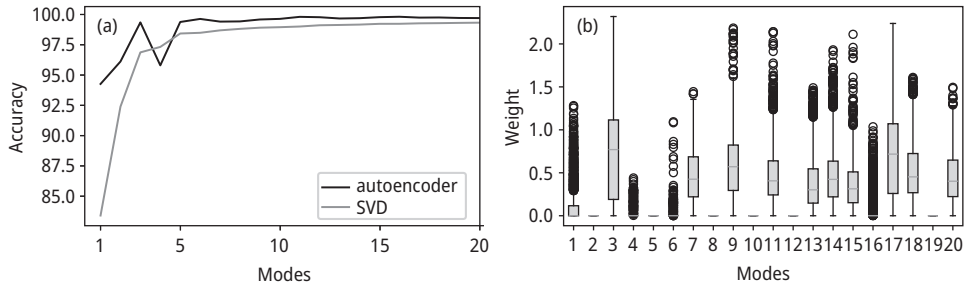


Figure 15.27 (a) Accuracy as a function of the number of SVD modes (grey) and latent space AE nodes trained over 100 epochs (black). The nonlinear curve fitting of the AE generally provides a more accurate representation of the data matrix with the same number of modes. (b) A box plot showing the projection of the 2414 images onto the latent space of the autoencoder with dimension $r = 20$.

15.11 Shallow Recurrent Decoder (SHRED) and Nonlinear SVD

Autoencoders provide a modern and exceptional architecture for discovering low-rank structure in data. Indeed, it provides a generalization of the standard SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$. Often such decompositions are used for spatio-temporal data in order to extract spatial features and their time dynamics. This is illustrated in Fig. 15.9, for instance. The DMD explicitly constructs a low-rank representation whose dynamics are linear and thus represented by exponentials. Yet another generalization of the SVD is the *SHallow REcurrent Decoder* (SHRED) deep learning model [124–126]. Although the focus here will be on spatial–temporal decomposition, SHRED can be more broadly applied to any data with multiple independent variables like time and space.

For spatio-temporal data, the goal in low-rank reduction is to produce dominant spatial and temporal features, which can be used to approximate a data matrix. To be more precise, the decomposition can be made more explicit by considering the decomposition for a single spatial dimension

$$\mathbf{A} = \hat{\mathbf{U}}(x)\hat{\mathbf{\Sigma}}\mathbf{V}^*(t)\,. \tag{15.11.1}$$

where the low-rank spatial features are explicitly encoded in $\hat{\mathbf{U}}(x)$ and the associated temporal evolution is given by $\mathbf{V}^*(t)$. DMD takes this one step further by assuming that $\mathbf{V}^*(t)$ has linear, exponential behaviour. Thus, a DMD approximation to the data takes the form

$$\mathbf{A} = \sum_{k=1}^{r} b_n \boldsymbol{\psi}_n(x)\exp(\omega_n t). \tag{15.11.2}$$

In both cases, the underlying decomposition takes the form of a separation of variables $\mathbf{A} = \mathbf{X}(x)\mathbf{T}(t)$. Indeed separation of variables is perhaps the most important solution technique, analytically and computationally, for solving partial differential equations. The entire

methodology of PDE solution techniques of previous chapters relies on a separation of variables structure. Note that separation of variables *does not* mean that the time and space variables are independent. Rather, they are connected, for instance, in separation of variables for PDEs by a constant, which turns out to be an eigenvalue. This is important as the SHRED separation of variables architecture retains a connection between time and space, as it is jointly trained.

SHRED is like the separation of variables method, but generalized to nonlinear (neural network) approximations. A compact representation of the SHRED algorithm, as shown in Fig. 15.28 is as two neural networks jointly trained to the form

$$\mathbf{A} = \mathbf{X}(x)\mathbf{T}(t) \rightarrow \mathbf{A} = f_{\boldsymbol{\theta}_1}(x) \circ g_{\boldsymbol{\theta}_2}(t) \tag{16.11.3}$$

where $g_{\boldsymbol{\theta}_2}(t)$ is trained largely to encode sequential (time) data and $f_{\boldsymbol{\theta}_1}(x)$ is largely responsible for mapping from the latent space of the sequential encoding to the full state-space. Of course, the time and space encoded are not independent, as they are jointly trained, and the decoder is dependent on the time history of the sequence model. This is also the case with the SVD (15.11.1) and DMD (15.11.2) models in which time and space are jointly regressed on.

Traditional time sequence models are built around *recurrent neural networks* (RNNs) [120; 127], which in turn can are very closely related to ideas in dynamical systems and numerical integration schemes. Specifically, the RNN looks to build a flow map (or numerical stepper) where

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \boldsymbol{\mu}) \tag{16.11.4}$$

which maps a solution one step forward in time from $\mathbf{x}_k = \mathbf{x}(t_k)$ to $\mathbf{x}_{k+1} = \mathbf{x}(t_{k+1})$. In Ch. 3, it was already shown how to construct a neural network map that accomplishes this task. Such a network took the form

$$\mathbf{x}_{k+1} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}_k) \tag{16.11.5}$$

where $\boldsymbol{\theta}$ are the network weights. An RNN trains over a sequence of p temporal data points, or evaluations of the variable of interest,

$$\mathbf{x}_{k+p} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{f}_{\boldsymbol{\theta}}(\cdots \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}_k)\cdots)). \tag{16.11.6}$$

This expression thus encodes a trajectory of information over p steps from $\mathbf{x}_k = \mathbf{x}(t_k)$ to $\mathbf{x}_{k+p} = \mathbf{x}(t_{k+p})$. This is referred to as the *unfolding* of the RNN as shown in Fig. 15.28. The decomposition advocated here uses the RNN structure for learning a sequence. So the SHRED model encodes data where one of the variables is a sequence. Such sequences, be it space or time, are typical in physics-based models, where the sequential data is enforced by relationships in the governing equations.

The RNN has another important feature, it can instead build a mapping of the original input variable $\mathbf{x}_k$ into a latent space representation. The latent space is often the more

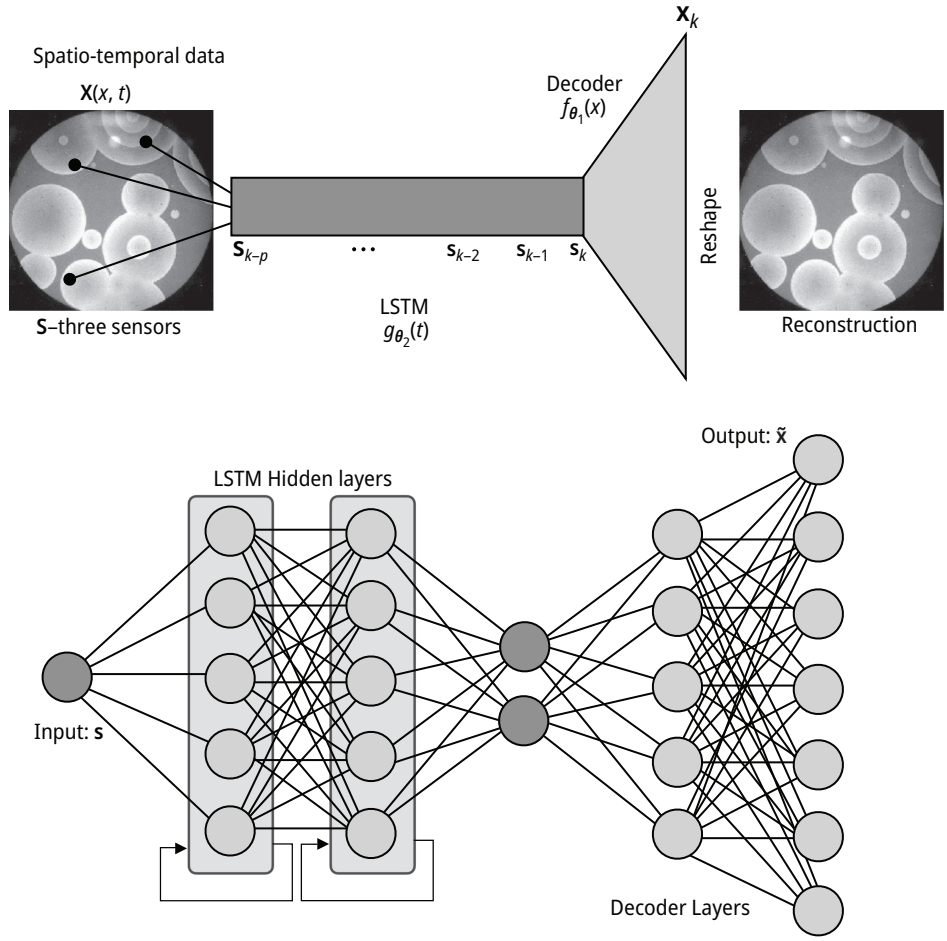


Figure 15.28 SHRED architecture that integrates an RNN and shallow decoder. The RNN structure trains on sequences of input data $\mathbf{s}_k = \mathbf{s}(t_k)$, which measure the spatio-temporal system at three randomly selected locations. The latent space of the RNN is the input to the decoder. Unlike a feed-forward neural network, the time history of the sequence, or memory, is used to train the neural network where the hidden layers are fed back into themselves upon training. Thus, the output of the neural network is fed back into the latent layers $\mathbf{f}_{\boldsymbol{\theta}}$. The top shows the conception, while the bottom shows the neural network architecture.

important variable for learning a recurrence map. If $\mathbf{h}_k = \mathbf{h}(t_k)$ is the latent representation, then the model becomes

$$\mathbf{h}_{k+1} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{h}_k, \mathbf{x}_{k+1}) \tag{16.11.7}$$

where the neural network is now dependent on the input variable. In either case, a neural network $\mathbf{f}_{\theta}$ is trained to advance the solution in time by training on trajectories from time

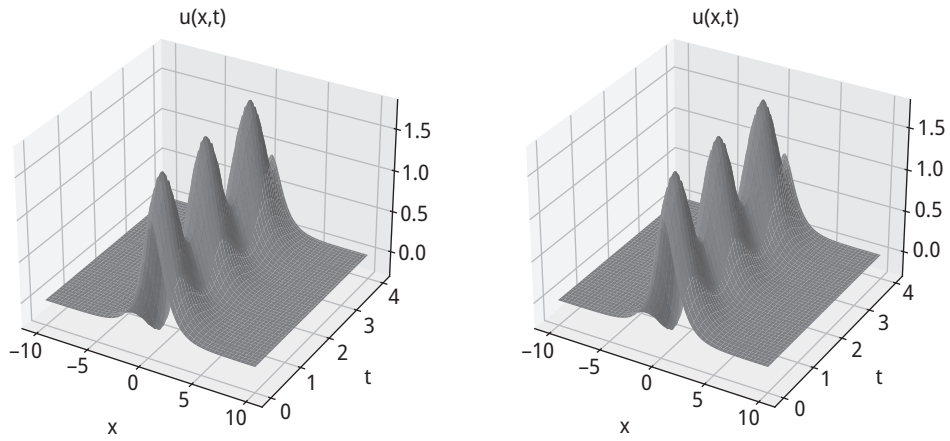


Figure 15.29 The left panel shows the true function $u(x,t) = [1 - 0.5\cos 2t]\mathrm{sech}x + [1 - 0.5\sin 2t]\mathrm{sech}x \tanh x$, which was example 2 (15.5.11). The right panel shows the reconstruction of the function using the SHRED architecture. Once trained, only the three time-series measurements s need to be retained for reconstruction.

t_1 to t_p. Importantly, the map is built from trajectory (sequence) information unlike the one-step mappings for learning time-stepping algorithms. Training over trajectories allows for the history (memory) of the solution to shape the neural network model.

The first RNNs were proposed in the 1980s with the foundational work of Rumelhart et al. [120] and Hopfield [127]. RNNs in the form of *long short-term memory* (LSTM) networks [128] became especially transformative in speech recognition applications since LSTMs, through its filtering architecture, regularize RNNs to avoid the vanishing gradient problem that is typically encountered in training RNNs. Other RNN architectures that have been constructed in order to avoid the vanishing or exploding gradients problem include *gated recurrent units* (GRU) [129], *echo state networks* (ESN) [130], and attention-based [1] transformer networks [131; 132]. Thus, LSTM, GRU, transformers, and ESN, along with its variants, are commonly used with time-series data.

In Fig. 15.29, we demonstrate the SHRED architecture using an LSTM for the recurrent, time sequence learning with a shallow decoder for reconstruction of the data from limited measurements [133]. The architecture is simple to train, in terms of limited data, minimal hyper-parameter tuning, model size, and low training times. Its performance is also quite remarkable as only three measurements, randomly selected [124], are required to construct the model. Three measurements are typically required in order to correctly *triangulate*, or disambiguate, the spatio-temporal data, much like the requirements for triangulation for uniquely determining position in cell phone applications. Unlike SVD and DMD, SHRED does not give an explicit representation of the time and space features. They are encoded implicitly in the neural network architecture.

16 Independent Component Analysis

16.1 The Concept of Independent Components 462
16.1.1 Image separation and SVD 465
16.1.2 SVD method for ICA 467
16.2 Image Separation Problem 469
16.2.1 Step 1: Rotation of the parallelogram 469
16.2.2 Step 2: Scaling of the parallelogram 471
16.2.3 Step 3: Rotation to produce a separable probability distribution 471
16.2.4 Completing the analysis 474
16.3 Image Separation and Python 475
16.3.1 Image statistics and the SVD 477
16.3.1.1 Step 1: Maximal/minimal variance angle detection 477
16.3.1.2 Step 2: Scaling of the principal components 478
16.3.1.3 Step 3: Rotation to separability 478
16.3.2 Image separation from reflection 479
16.4 FastICA Algorithm 480
16.4.1 The FastICA Algorithm 482
16.4.2 Python Implementation of FastICA 482

The concept of principal components or of a proper orthogonal decomposition are fundamental to data analysis. Essentially, there is an assertion, or perhaps a hope, that underlying seemingly complex and unordered data, there is, in fact, some characteristic, and perhaps low-dimensional ordering of the data. The singular value decomposition of a matrix, although a mathematical idea, was actually a formative tool for the interpretation and ordering of the data. In this chapter on *independent component analysis* (ICA), the ideas turn to data sets for which there is more than one statistically independent object in the data. ICA provides a foundational mathematical method for extracting interleaved data sets in a fairly straightforward way. Its applications are wide and varied, but our primary focus will be application of ICA in the context of image analysis.

16.1 The Concept of Independent Components

Independent component analysis (ICA) is a powerful tool that extends the concepts of PCA, POD, and SVD. Moreover, you are already intuitively familiar with the concept as some of

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0017

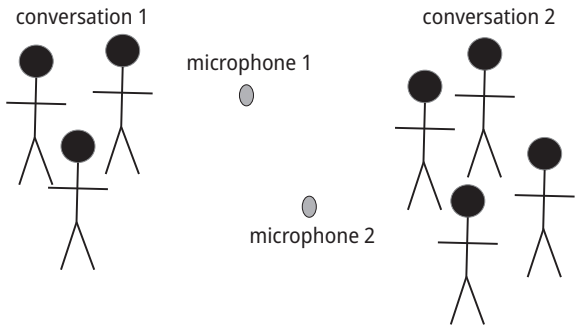


Figure 16.1 Envisioned cocktail party problem. Two groups of people are having separate conversations, which are recorded from microphone 1 and 2. The signals from the two conversations are mixed together at microphone 1 and 2 according to the placement relative to the conversations. From the two microphone recordings, the individual conversations can then be extracted.

the examples here will show. A simple way to motivate thinking about ICA is by considering the *cocktail party problem*. Thus, consider two conversations in a room that are happening simultaneously. How is it that the two different acoustic signals of conversation one and two can be separated out? Figure 16.1 depicts a scenario where two groups are conversing. Two microphones are placed in the room at different spatial locations and from the two signals $s_1(t)$ and $s_2(t)$ a mathematical attempt is made to separate the signals that have been mixed at each of the microphone locations. Provided that the noise level is not too large or that the conversation volumes are sufficiently large, humans can perform this task with remarkable ease. In our case, the two microphones are our two ears. Indeed, this scenario and its mathematical foundations are foundational to the concept of eavesdropping on a conversation.

From a mathematical standpoint, this problem can be formulated with the following mixing equation:

$$x_1(t) = a_{11}s_1 + a_{12}s_2 \tag{16.1.1a}$$

$$x_2(t) = a_{21}s_1 + a_{22}s_2 \tag{16.1.1b}$$

where $x_1(t)$ and $x_2(t)$ are the mixed, recorded signals at microphones 1 and 2, respectively. The coefficients a_{ij} are the mixing parameters that are determined by a variety of factors, including the placement of the microphones in the room, the distance to the conversations, and the overall room acoustics. Note that we are omitting time-delay signals that may reflect off the walls of the room. This problem also resembles quite closely what may happen in a large number of applications. For instance, consider the following:

- **Radar detection** If there are numerous targets that are being tracked, then there is significant mixing in the scattered signal from all of the targets. Without a method for clear separation of the targets, the detector becomes useless for identifying location.

- **Electroencephalogram (EEG)** EEG readings are electrical recordings of brain activities. Typically, these EEG readings are from multiple locations of the scalp. However, at each EEG reading location, all the brain activity signals are mixed, thus preventing a clear understanding of how many underlying signals are contained within. With a large number of EEG probes, ICA allows for the separation of the brain activity readings and a better assessment of the overall neural activity.
- **Terminator salvation** If you remember in the movie, John Connor and the resistance found a hidden signal (ICA) embedded in the normal signals in the communications sent between the Terminators and Skynet vehicles and ships. Although not mentioned in the movie, this was clearly somebody in the future who is reading this book now. Somehow that bit of sweet math only made it to the cutting room floor. Regardless, one should not underestimate how awesome data analysis skills are in the real world of the future.

The ICA method is closely related to the *blind source separation* (BSS) (or blind signal separation) method. Here, the sources refer to the original signals, or independent components, and blind refers to the fact that the mixing matrix coefficients a_{ij} are unknown.

A more formal mathematical framework for ICA can be established by considering the following: *Given N distinct linear combinations of N signals (or data sets), determine the original N images*. This is the succinct statement of the mathematical objective of ICA. Thus, to generalize (16.1.1b) to the N-dimensional system, we have

$$x_j(t) = a_{j1}s_1 + a_{j2}s_2 + \cdots + a_{jN}s_N \quad 1 \le j \le N. \tag{16.1.2}$$

Expressed in matrix form, this results in

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{16.1.3}$$

where $\mathbf{x}$ are the mixed signal measurements, $\mathbf{s}$ are the original signals we are trying to determine, and $\mathbf{A}$ is the matrix of coefficients that determines how the signals are mixed as a function of the physical system of interest. At first, one might simply say that the solution to this is trivial, and it is given by

$$\mathbf{s} = \mathbf{A}^{-1}\mathbf{x} \tag{16.1.4}$$

where we are assuming that the placement of our measurement devices is such that $\mathbf{A}$ is nonsingular and its inverse exists. Such a solution makes the following assumption: we know the matrix coefficients a_{ij}. The point is, we do not know them. Nor do we know the signal $\mathbf{s}$. Mathematically then, the aim of ICA is to approximate the coefficients of the matrix $\mathbf{A}$, and in turn the original signals $\mathbf{s}$, under as general assumptions as possible.

The statistical model given by (16.1.3) is called *independent component analysis*. ICA is a *generative model*, which means that it describes how the observed data are generated by a process of mixing in the components $s_j(t)$. The independent components are *latent variables*, meaning they are never directly observed. The key step in the ICA models is to assume the

following: *the underlying signals $s_j(t)$ are statistically independent with probability distributions that are not Gaussian*. Statistical independence and higher-order moments of the probability distribution (thus requiring non-Gaussian distributions) are the critical mathematical tools that will allow us to recover the signals.

16.1.1 Image separation and SVD

To make explicit the mathematical methodology to be pursued here, a specific example of image separation will be used. Although there are a variety of mathematical alternatives for separating the independent components [134], the approach considered here will be based upon PCA and SVD. To illustrate the concept of ICA, consider the example data represented in Fig. 16.2. The three panels are fundamental to understanding the concept of ICA. In the left panel (a), measurements are taken of a given system and are shown to project nicely onto a dominant direction whose leading principal component is denoted by the red vector. This red vector would be the principal component with a length σ_1 determined by the largest singular value. It is clear that the singular value σ_2 corresponding to the orthogonal direction of the second principal component would be considerably smaller. In the middle panel (b), the measurements indicate that there are two principal directions in the data fluctuations. If an SVD is applied directly to the data, then the dominant singular direction would be approximately the green vector. Yet it is clear that the green vector does not accurately represent the data. Rather, the data seems to be organized *independently* along two different directions. An SVD of the two independent data sets would produce two principal components, i.e. two *independent component analysis* vectors. Thus, it is critical to establish a method for separating out data of this sort into their independent directions. Finally, in the third panel (c), Gaussian distributed data is shown, where no principal components can be identified with certainty. Indeed, there are an infinite number of possible orthogonal projections that can be made. Two arbitrary directions have been shown in panel (c). This graphic shows why Gaussian distributed random variables are disastrous for ICA; no projections onto the independent components can be accomplished.

We now turn our attention to the issue of image separation. Figure 16.3 demonstrates a prototypical example of what can occur when photographing an image behind glass. The

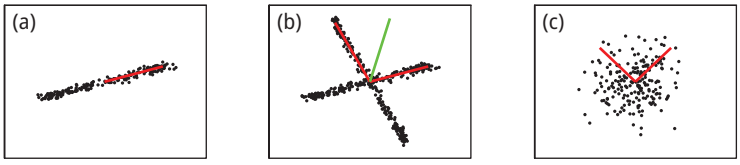


Figure 16.2 Illustration of the principals of PCA (a), ICA (b), and the failure of Gaussian distributions to distinguish principal components (c). The red vectors show the principal directions, while the green vector in the middle panel shows what would be the principal direction if a direct SVD where applied rather than an ICA. The principal directions in (c) are arbitrarily chosen since no principal directions can be distinguished.

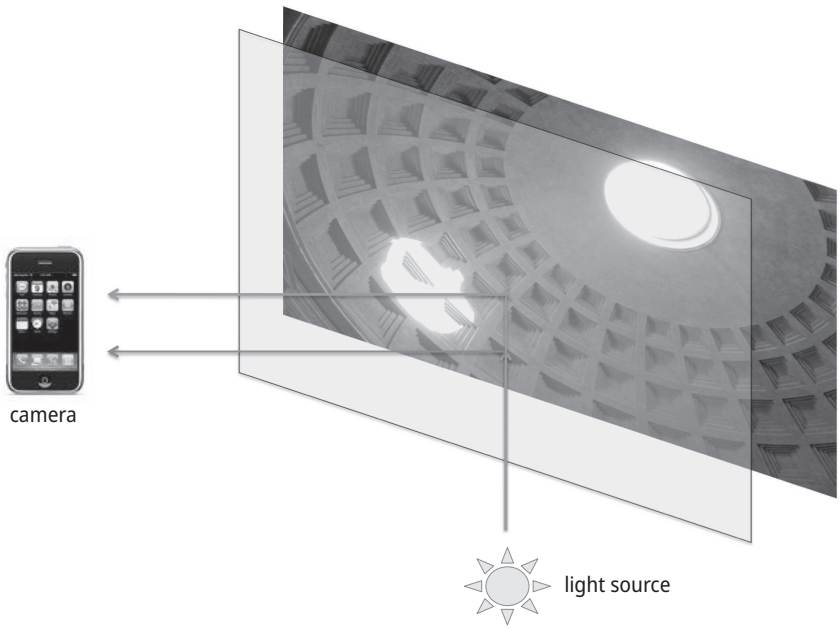


Figure 16.3 Illustration of image separation problem. A piece of glass reflects part of the background light source, while some of the light penetrates to the image and is reflected to the camera. Thus, the camera sees two images: the image (painting) and the reflection of the background light source.

glass encasing of the artwork, or image, produces a reflection of the backlit source along with the image itself. Amazingly, the human eye can very easily compensate for this and *see through* much of the reflection to the original image. Alternatively, if we choose, we can also focus on the reflected image of the backlit source. Ultimately, there is some interest in training a computer to do this image processing automatically. Indeed, in artificial vision, algorithms such as these are important for mimicking natural human behaviours and abilities.

The light reflected off the glass is partially polarized and provides the crucial degree of freedom necessary for our purposes. Specifically, to separate the images, a photo is taken as illustrated in Fig. 16.3 but now with a linear polarizer placed in front of the camera lens. Two photos are taken where the linear polarizer is rotated 90 degrees from each other. This is a simple home experiment essentially. But you must have some control over the focus of your camera in terms of focusing and flash. The picture to be considered is hanging in my office: **The Judgement of Paris** by John Flaxman (1755–1826). Flaxman was an English sculptor, draughtsman, and illustrator who enjoyed a long and brilliant career that included a stint at Wedgwood in England. In fact, at the age of 19, Josiah Wedgwood hired Flaxman as a modeller of classic and domestic friezes, plaques, ornamental vessels, and medallion portraits. You can still find many Wedgwood ornamental vessels today that have signature Flaxman works moulded on their sides. It was during this period that the manufacturers of that time perfected their style and earned great reputations. But what gained Flaxman his

general fame was not his work in sculpture proper, but his designs and engravings for the **Iliad**, **Odyssey**, and **Dante's Inferno**. All of his works on these mythological subjects were engraved by Piroli with considerable loss to Flaxman's own lines. The greatest collection of Flaxman's work is housed in the Flaxman Gallery at the University College London.

In the specific Flaxman example considered here, one of the most famous scenes of mythology is depicted: **The Judgement of Paris**. In this scene, Hermes watches as young Paris plays the part of judge. His task is to contemplate giving the golden apple with the inscription *To the Fairest* to either Venus, Athena, or Hera. This golden apple was placed at the wedding reception of Peleus and Thetis, father and mother of peerless Achilles, by Discord since she did not receive a wedding invitation. Each goddess in turn promises him something for the apple: Athena promises that he will become the greatest warrior in history, Juno promises him to be greatest ruler in the land, ruling over a vast and unmatched empire. Last comes Venus who promises him that the most beautiful woman in the world will be his (Helen of Troy). And as a result we have the Trojan War for which Helen was fought for. Figures 16.4 and 16.5 represent this Flaxman scene along with the reflected image of my office window. The two pictures are taken with a linear polarizer in front of a camera that has been rotated by 90 degrees between the photos.

16.1.2 SVD method for ICA

Given the physical problem, how is SVD used then to separate the two images observed in Figs. 16.4 and 16.5? Consider the action of the SVD on the image mixing matrix $\mathbf{A}$ of (16.1.3). In this case, there are two images $\mathbf{S}_1$ and $\mathbf{S}_2$ that we are working with. This gives us



Figure 16.4 The Judgement of Paris by John Flaxman (1755–1826). This is a plate made by Flaxman for an illustrated version of Homer's Iliad. The picture was taken in my office, and so you can see both a faint reflection of my books as well as strong signature of my window which overlooks Drumheller Fountain.



Figure 16.5 This is an identical picture to Fig. 16.4 with a linear polarizer rotated 90 degrees. The polarization state of the reflected field is affected by the linear polarizer, thus allowing us to separate the images.

six unknowns ($\mathbf{S}_1, \mathbf{S}_2, a_{11}, a_{12}, a_{21}, a_{22}$) with only the two constraints from (16.1.3). Thus, the system cannot be solved without further assumptions being made. The first assumption will be that the two images are statistically independent. If the pixel intensities are denoted by $\mathbf{S}_1$ and $\mathbf{S}_2$, then statistical independence is mathematically expressed as

$$P(\mathbf{S}_1, \mathbf{S}_2) = P(\mathbf{S}_1)P(\mathbf{S}_2) \tag{16.1.5}$$

which states that the joint probability density is separable. No specific form is placed upon the probability densities themselves aside from this: they cannot be Gaussian distributed. In practice, this is a reasonable constraint since natural images are rarely Gaussian. A second assumption is that the matrix $\mathbf{A}$ is full rank, thus assuming that the two measurements of the image have indeed been filtered with different polarization states. These assumptions yield an estimation problem which is the ICA.

Consider what happens under the SVD process to the matrix $\mathbf{A}$

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{16.1.6}$$

where again $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices that simply lead to rotation and $\mathbf{\Sigma}$ stretches (scales) an image as prescribed by the singular values. A graphical illustration of this process is shown in Fig. 16.6 for distributions that are uniform, i.e. they form a square. The mixing matrix $\mathbf{A}$ can be thought to first rotate the square via unitary matrix $\mathbf{V}^*$, then stretch it into a parallelogram via the diagonal matrix $\mathbf{\Sigma}$, and then rotate the parallelogram via the unitary matrix $\mathbf{U}$. This is now the mixed image $\mathbf{X}$. *The estimation, or ICA, of the independent images thus reduces to finding how to transform the rotated parallelogram back into a square. Or mathematically, transforming the mixed image back into a separable product of one-dimensional*

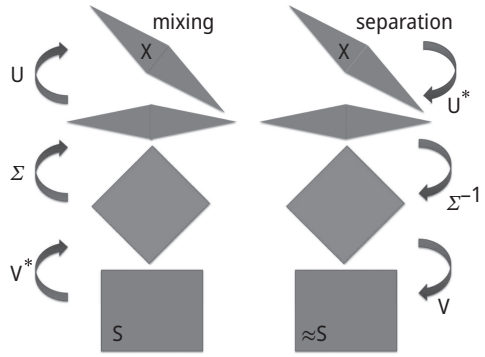


Figure 16.6 Graphical depiction of the SVD process of mixing the two images. The reconstruction of the images is accomplished by approximating the inversions of the SVD matrices so as to achieve a separable (statistically independent) probability distribution of the two images.

probability distributions. This is the defining mathematical goal of the ICA image analysis problem, or any general ICA reduction technique.

16.2 Image Separation Problem

The method proposed here to separate two images relies on reversing the action of the SVD on the two statistically independent images. Again, the matrix $\mathbf{A}$ in (16.1.6) is not known, so a direct computation of the SVD cannot be performed. However, each of the individual matrices can be approximated by considering its net effect on the assumed uniformly distributed images.

Three specific computations must be made: (i) the rotation of the parallelogram must be approximated, (ii) the scaling of the parallelogram according to the variances must be computed, and (iii) the final rotation back to a separable probability distribution must be obtained. Each step is outlined below.

16.2.1 Step 1: Rotation of the parallelogram

To begin, consider once again Fig. 16.6. Our first objective is to undo the rotation of the unitary matrix $\mathbf{U}$. Thus, we will ultimately want to compute the inverse of this matrix, which is simply $\mathbf{U}^*$. In a geometrical way of thinking, our objective is to align the long and short axes of the parallelogram with the primary axis, as depicted in the two top right shaded boxes of Fig. 16.6. The angle of the parallelogram relative to the primary axes will be denoted by θ, and the long and short axes correspond to the axes of the maximal and minimal variance, respectively. From the image data itself, then, the maximal and minimal variance directions will be extracted. Assuming mean-zero measurements, the variance at an arbitrary angle of orientation is given by

$$Var(\theta) = \sum_{j=1}^{N} \left\{ [x_1(j) \quad x_2(j)] \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \right\}^2. \tag{16.2.1}$$

The maximal variance is determined by computing the angle θ that maximizes this function. It will be assumed that the corresponding angle of minimal variance will be orthogonal to this at $\theta - \pi/2$. These axes are essentially the principal component directions that would be computed if we actually knew components of the matrix $\mathbf{A}$.

The maximum of (16.2.1) with respect to θ can be found by differentiating $Var(\theta)$ and setting it equal to zero. To do this, (16.2.1) is rewritten as

$$\begin{aligned} Var(\theta) &= \sum_{j=1}^{N} \left\{ [x_1(j) \quad x_2(j)] \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \right\}^2 \\ &= \sum_{j=1}^{N} [x_1(j)\cos\theta + x_2(j)\sin\theta]^2 \\ &= \sum_{j=1}^{N} x_1^2(j)\cos^2\theta + 2x_1(j)x_2(j)\cos\theta\sin\theta + x_2^2(j)\sin^2\theta. \end{aligned} \tag{16.2.2}$$

Differentiating with respect to θ then gives

$$\begin{aligned} \frac{d}{d\theta}Var(\theta) &= 2\sum_{j=1}^{N} -x_1^2(j)\sin\theta\cos\theta + x_1(j)x_2(j)\left[\cos^2\theta - \sin^2\theta\right] + x_2^2(j)\sin\theta\cos\theta \\ &= 2\sum_{j=1}^{N} \left[x_2^2(j) - x_1^2(j)\right]\sin\theta\cos\theta + x_1(j)x_2(j)\left[\cos^2\theta - \sin^2\theta\right] \\ &= \sum_{j=1}^{N} \left[x_2^2(j) - x_1^2(j)\right]\sin 2\theta + 2x_1(j)x_2(j)\cos 2\theta \end{aligned} \tag{16.2.3}$$

which upon setting this equal to zero gives the value of θ desired. Thus, taking $d(Var(\theta))/d\theta = 0$ gives

$$\frac{\sin 2\theta}{\cos 2\theta} = \frac{-2\sum_{j=1}^{N} x_1(j)x_2(j)}{\sum_{j=1}^{N} x_2^2(j) - x_1^2(j)}, \tag{16.2.4}$$

or in terms of θ alone

$$\theta_0 = \frac{1}{2}\tan^{-1}\left[\frac{-2\sum_{j=1}^{N} x_1(j)x_2(j)}{\sum_{j=1}^{N} x_2^2(j) - x_1^2(j)}\right]. \tag{16.2.5}$$

In the polar coordinates $x_1(j) = r(j)\cos\psi(j)$ and $x_2(j) = r(j)\sin\psi(j)$, the expression reduces further to

$$\theta_0 = \frac{1}{2}\tan^{-1}\left[\frac{\sum_{j=1}^{N} r^2(j)\sin 2\psi(j)}{\sum_{j=1}^{N} r^2(j)\cos 2\psi(j)}\right]. \tag{16.2.6}$$

The rotation matrix, or unitary transformation, associated with the rotation of the parallelogram back to its aligned position is then

$$\mathbf{U}^* = \begin{bmatrix} \cos\theta_0 & \sin\theta_0 \\ -\sin\theta_0 & \cos\theta_0 \end{bmatrix} \tag{16.2.7}$$

with the angle θ_0 computed directly from the experimental data.

16.2.2 Step 2: Scaling of the parallelogram

The second task is to undo the principal component scaling achieved by the singular values of the SVD decomposition. This process is illustrated as the second step in the right column of Fig. 16.6. This task, however, is rendered straightforward now that the principal axes have been determined from step 1. In particular, the assumption was that along the direction θ_0, the maximal variance is achieved, while along $\theta_0 - \pi/2$, the minimal variance is achieved. Thus, the components, or singular values, of the diagonal matrix $\mathbf{\Sigma}^{-1}$ can be computed.

The variances along the two principal component axes are given by

$$\sigma_1 = \sum_{j=1}^{N} \left\{ [x_1(j) \;\; x_2(j)] \begin{bmatrix} \cos\theta_0 \\ \sin\theta_0 \end{bmatrix} \right\}^2 \tag{16.2.8a}$$

$$\sigma_2 = \sum_{j=1}^{N} \left\{ [x_1(j) \;\; x_2(j)] \begin{bmatrix} \cos(\theta_0 - \pi/2) \\ \sin(\theta_0 - \pi/2) \end{bmatrix} \right\}^2 . \tag{16.2.8b}$$

This then gives the diagonal elements of the matrix $\mathbf{\Sigma}$. To undo this scaling, the inverse of $\mathbf{\Sigma}$ is constructed so that

$$\mathbf{\Sigma}^{-1} = \begin{bmatrix} 1/\sqrt{\sigma_1} & 0 \\ 0 & 1/\sqrt{\sigma_2} \end{bmatrix}. \tag{16.2.9}$$

This matrix, in combination with that of step 1, easily undoes the principal component direction of rotation and its associated scaling. However, this process has only decorrelated the images, and a separable probability distribution has not yet been produced.

16.2.3 Step 3: Rotation to produce a separable probability distribution

The final rotation to separate the probability distributions is a more subtle and refined issue, but critical to producing nearly separable probability distributions. This separation process typically relies on the higher moments of the probability distribution. Since the mean has been assumed to be zero, and there is no reason to believe that there is an asymmetry in the probability distributions, i.e. higher-order odd moments (such as the skewness) are negligible, the next dominant statistical moment to consider is the fourth moment, or the kurtosis of the probability distribution. The goal will be to minimize this fourth-order moment, and by doing so we will determine the appropriate rotation angle. Note that the second moment

has already been handled through steps 1 and 2. Said in a different mathematical way, minimizing the kurtosis will be another step in trying to approximate the probability distribution of the images as separable functions so that $P(\mathbf{S}_1\mathbf{S}_2) \approx P(\mathbf{S}_1)P(\mathbf{S}_2)$.

The kurtosis of a probability distribution is given by

$$kurt(\phi) = K(\phi) = \sum_{j=1}^{N} \left\{ [\bar{x}_1(j) \quad \bar{x}_2(j)] \begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix} \right\}^4 \tag{16.2.10}$$

where ϕ is the angle of rotation associated with the unitary matrix $\mathbf{U}$, and the variables $\bar{x}_1(j)$ and $\bar{x}_2(j)$ represent the image that has undergone the two steps of transformation as outlined previously.

For analytic convenience, a normalized version of the above definition of kurtosis will be considered. Specifically, a normalized version is computed in practice. The expedience of the normalization will become clear through the algebraic manipulations. Thus, consider

$$\bar{K}(\phi) = \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j) + \bar{x}_2^2(j)} \left\{ [\bar{x}_1(j) \quad \bar{x}_2(j)] \begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix} \right\}^4. \tag{16.2.11}$$

As in our calculation regarding the second moment, the kurtosis will be written in a more natural form for differentiation

$$\begin{aligned}
\bar{K}(\phi) &= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j) + \bar{x}_2^2(j)} \left\{ [\bar{x}_1(j) \quad \bar{x}_2(j)] \begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix} \right\}^4 \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j) + \bar{x}_2^2(j)} \left[\bar{x}_1(j)\cos\phi + \bar{x}_2(j)\sin\phi \right]^4 \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j) + \bar{x}_2^2(j)} \left[\bar{x}_1^4(j)\cos^4\phi + 4\bar{x}_1^3(j)\bar{x}_2(j)\cos^3\phi\sin\phi \right. \\
&\quad \left. + 6\bar{x}_1^2(j)\bar{x}_2^2(j)\cos^2\phi\sin^2\phi + 4\bar{x}_1(j)\bar{x}_2^3(j)\cos\phi\sin^3\phi + \bar{x}_2^4(j)\sin^4\phi \right] \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j) + \bar{x}_2^2(j)} \left[\frac{1}{8}\bar{x}_1^4(j)(3 + 4\cos 2\phi + \cos 4\phi) \right. \\
&\quad + \bar{x}_1^3(j)\bar{x}_2(j)(\sin 2\phi + (1/2)\sin 4\phi) \\
&\quad + \frac{3}{4}\bar{x}_1^2(j)\bar{x}_2^2(j)(1 - \cos 4\phi) + \bar{x}_1(j)\bar{x}_2^3(j)(\sin 2\phi - (1/2)\sin 4\phi) \\
&\quad \left. + \frac{1}{8}\bar{x}_2^4(j)(3 - 4\cos 2\phi + \cos 4\phi) \right].
\end{aligned} \tag{16.2.12}$$

This quantity needs to be minimized with an appropriate choice of ϕ. Thus, the derivative $d\bar{K}/d\phi$ must be computed and set to zero:

$$\begin{aligned}
\frac{d\bar{K}(\phi)}{d\phi} &= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j)+\bar{x}_2^2(j)} \Big[\frac{1}{8}\bar{x}_1^4(j)(-8\sin 2\phi - 4\sin 4\phi) \\
&\quad + \bar{x}_1^3(j)\bar{x}_2(j)(2\cos 2\phi + 2\cos 4\phi) \\
&\quad + \ 3\bar{x}_1^2(j)\bar{x}_2^2(j)\sin 4\phi + \bar{x}_1(j)\bar{x}_2^3(j)(2\cos 2\phi - 2\cos 4\phi) \\
&\quad + \frac{1}{8}\bar{x}_2^4(j)(8\sin 2\phi - 4\sin 4\phi)\Big] \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j)+\bar{x}_2^2(j)} \Big\{ \big[\bar{x}_2^4(j) - \bar{x}_1^4(j)\big]\sin 2\phi \\
&\quad + \big[2\bar{x}_1(j)\bar{x}_2(j)^3 + 2\bar{x}_1^3(j)\bar{x}_2(j)\big]\cos 2\phi \\
&\quad + \big[2\bar{x}_1^3(j)\bar{x}_2(j) - 2\bar{x}_1(j)\bar{x}_2^3(j)\big]\cos 4\psi \\
&\quad + \big[3\bar{x}_1(j)^2\bar{x}_2^2(j) - (1/2)\bar{x}_1^4(j) - (1/2)\bar{x}_2^4(j)\big]\sin 4\psi \Big\} \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j)+\bar{x}_2^2(j)} \Big\{ \big[(\bar{x}_1^2(j)+\bar{x}_2^2(j))(\bar{x}_2^2(j)-\bar{x}_1^2(j))\big]\sin 2\phi \\
&\quad + \big[2\bar{x}_1(j)\bar{x}_2(j)(\bar{x}_1^2(j)+\bar{x}_2^2(j))\big]\cos 2\phi \\
&\quad + \big[2\bar{x}_1^3(j)\bar{x}_2(j) - 2\bar{x}_1(j)\bar{x}_2^3(j)\big]\cos 4\psi \\
&\quad + \big[3\bar{x}_1(j)^2\bar{x}_2^2(j) - (1/2)\bar{x}_1^4(j) - (1/2)\bar{x}_2^4(j)\big]\sin 4\psi \Big\} \\
&= \sum_{j=1}^{N} \big[\bar{x}_2^2(j) - \bar{x}_1^2(j)\big]\sin 2\phi + 2\bar{x}_1(j)\bar{x}_2(j)\cos 2\phi \\
&\quad + \frac{1}{\bar{x}_1^2(j)+\bar{x}_2^2(j)} \Big\{ \big[2\bar{x}_1^3(j)\bar{x}_2(j) - 2\bar{x}_1(j)\bar{x}_2^3(j)\big]\cos 4\psi \\
&\quad + \big[3\bar{x}_1(j)^2\bar{x}_2^2(j) - (1/2)\bar{x}_1^4(j) - (1/2)\bar{x}_2^4(j)\big]\sin 4\psi \Big\} \\
&= \sum_{j=1}^{N} \frac{1}{\bar{x}_1^2(j)+\bar{x}_2^2(j)} \Big\{ \big[2\bar{x}_1^3(j)\bar{x}_2(j) - 2\bar{x}_1(j)\bar{x}_2^3(j)\big]\cos 4\psi \\
&\quad + \big[3\bar{x}_1(j)^2\bar{x}_2^2(j) - (1/2)\bar{x}_1^4(j) - (1/2)\bar{x}_2^4(j)\big]\sin 4\psi \Big\}, \qquad (16.2.13)
\end{aligned}$$

where the terms proportional to $\sin 2\phi$ and $\cos 2\phi$ add to zero since they are minimized when considering the second moment or variance, i.e. we would like to minimize both the variance and the kurtosis, and this calculation does both. By setting this to zero, the angle ϕ_0 can be determined to be

$$\phi_0 = \frac{1}{4}\tan^{-1}\left[\frac{-\sum_{j=1}^{N}\left[2\bar{x}_1^3(j)\bar{x}_2(j) - 2\bar{x}_1(j)\bar{x}_2^3(j)\right]/\left[\bar{x}_1^2(j)+\bar{x}_2^2(j)\right]}{\sum_{j=1}^{N}\left[3\bar{x}_1(j)^2\bar{x}_2^2(j) - (1/2)\bar{x}_1^4(j) - (1/2)\bar{x}_2^4(j)\right]/\left[\bar{x}_1^2(j)+\bar{x}_2^2(j)\right]}\right]. \tag{16.2.14}$$

When converted to polar coordinates, this reduces very nicely to the following expression:

$$\phi_0 = \frac{1}{4}\tan^{-1}\left[\frac{\sum_{j=1}^{N} r^2 \sin 4\psi(j)}{\sum_{j=1}^{N} r^2 \cos 4\psi(j)}\right]. \tag{16.2.15}$$

The rotation back to the approximately statistically independent square is then given by

$$\mathbf{V} = \begin{bmatrix} \cos\phi_0 & \sin\phi_0 \\ -\sin\phi_0 & \cos\phi_0 \end{bmatrix} \tag{16.2.16}$$

with the angle θ_0 computed directly from the experimental data. At this point, it is unknown whether the angle ϕ_0 is a minimum or maximum of the kurtosis, and this should be checked. Sometimes this is a maximum, especially in cases when one of the image histograms has long tails relative to the other [135].

16.2.4 Completing the analysis

Recall that our purpose was to compute, through the statistics of the images (pixels) themselves, the SVD decomposition. The method relied on computing (approximating) the principal axes and scaling of the principal components. The final piece was to try and make the probability distribution as separable as possible by minimizing the kurtosis as a function of the final rotation angle.

To reconstruct the image, the following linear algebra operation is performed:

$$\begin{aligned}\mathbf{s} &= \mathbf{A}^{-1}\mathbf{x} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^*\mathbf{x} \\ &= \begin{bmatrix} \cos\phi_0 & \sin\phi_0 \\ -\sin\phi_0 & \cos\phi_0 \end{bmatrix}\begin{bmatrix} 1/\sqrt{\sigma_1} & 0 \\ 0 & 1/\sqrt{\sigma_2} \end{bmatrix}\begin{bmatrix} \cos\theta_0 & \sin\theta_0 \\ -\sin\theta_0 & \cos\theta_0 \end{bmatrix}\mathbf{x}.\end{aligned} \tag{16.2.17}$$

A few things should be noted about the inherent ambiguities in the recovery of the two images. The first is the ordering ambiguity. Namely, the two matrices are indistinguishable:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{21} & a_{22} \\ a_{11} & a_{12} \end{bmatrix}\begin{bmatrix} x_2 \\ x_1 \end{bmatrix}. \tag{16.2.18}$$

Thus, images one and two are arbitrary to some extent. In practice, this does not matter since the aim was simply to separate, not label, the data measurements. There is also an ambiguity in the scaling since

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}/\alpha & a_{12}/\delta \\ a_{21}/\alpha & a_{22}/\delta \end{bmatrix}\begin{bmatrix} \alpha x_1 \\ \delta x_2 \end{bmatrix}. \tag{16.2.19}$$

Again, this does not matter since the two separated images can then be rescaled to their full intensity range afterwards. Regardless, this shows the basic process that needs to be applied to the system data in order to construct a self-consistent algorithm capable of image separation.

16.3 Image Separation and Python

Using Python, the previously discussed algorithm will be implemented. However, before proceeding to try and extract the images displayed in Figs. 16.4 and 16.5, a more idealized case is considered. Consider the two ideal images of Sicily in Fig. 16.7. These two images will be mixed with two different weights to produce two mixed images. Our objective will be to use the algorithm outlined in the previous section to separate out the images. Upon separation, the images can be compared with the ideal representation of Fig. 16.7. To load in the images and start the processing, the following Python commands are used.

```
S1 = np.array(Image.open('sicily1.jpg'))
S2 = np.array(Image.open('sicily2.jpg'))
```

The two images are quite different with one overlooking the Mediterranean from the medieval village of Erice on the northwest corner, and the second is of the Cappella Palatina in Palermo. Recall that what is required is statistical independence of the two images. Thus, the pixel strengths and colours should be largely decorrelated throughout the image.

The two ideal images are now mixed with the matrix $\mathbf{A}$ in (16.1.3). Specifically, we will consider the arbitrarily chosen mixing of the form

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 4/5 & \beta \\ 1/2 & 2/3 \end{bmatrix} \tag{16.3.1}$$

where in what follows, we will consider the values $\beta = 1/5, 3/5$. This mixing produces two composite images with the strengths of images 1 and 2 altered between the two composites. This can be easily accomplished in Python with the following commands.



Figure 16.7 Images 1 and 2 of Sicily: an ocean view from the hills of Erice and an interior view of the Cappella Palatina in Palermo.



Figure 16.8 The top two figures show the mixed images produced from (16.3.1) with $\beta = 1/5$. The bottom two figures are the reconstructed images from the image separation process and approximation to the SVD. For the value of $\beta = 1/5$, the Cappella Palatina has been exceptionally well reconstructed, while the view from Erice has quite a bit of residual from the second image.

```
A = np.array([[0.7, 0.1], [1/2, 2/3]])

X1 = A[0, 0] * S1 + A[0, 1] * S2
X2 = A[1, 0] * S1 + A[1, 1] * S2

plt.subplot(2, 2, 1)
plt.imshow(X1.astype('uint8'))
plt.subplot(2, 2, 2)
plt.imshow(X2.astype('uint8'))
```

Note that the coefficients of the mixing matrix $\mathbf{A}$ will have a profound impact on our ability to extract one image from another. Indeed, just switching the parameter β from $1/5$ to $3/5$ will show the impact of a small change to the mixing matrix. It is these images that we wish to reconstruct by numerically computing an approximation to the SVD. The top rows of Figs. 16.8 and 16.9 demonstrate the mixing that occurs with the two ideal images given the mixing matrix (16.3.1) with $\beta = 1/5$ (Fig. 16.8) and $\beta = 3/5$ (Fig. 16.9).



Figure 16.9 The top two figures show the mixed images produced from (16.3.1) with $\beta = 3/5$. The bottom two figures are the reconstructed images from the image separation process and approximation to the SVD. For the value of $\beta = 3/5$, the view from Erice has been has been well reconstructed aside from the top right corner of the image, while the Cappella Palatina remains of poor quality.

16.3.1 Image statistics and the SVD

The image separation method relies on the statistical properties of the image for reconstructing an approximation to the SVD decomposition. Three steps have been outlined in the previous section that must be followed: first the rotation of the parallelogram must be computed by finding the maximal and minimal directions of the variance of the data. Second, the scaling of the principal component directions is evaluated by calculating the variance, and third, the final rotation is computed by minimizing both the variance and kurtosis of the data. This yields an approximately separable probability distribution. The three steps are each handled in turn.

16.3.1.1 Step 1: Maximal/minimal variance angle detection

This step is illustrated in the top right of Fig. 16.6. The actual calculation that must be performed for calculating the rotation angle is given by either (16.2.5) or (16.2.6). Once computed, the desired rotation matrix $\mathbf{U}^*$ given by (16.2.7) can be constructed. Recall the underlying assumption that a mean-zero distribution is considered. In Python, this computation takes the following form.

```
X1bw = rgb2gray(X1)
X2bw = rgb2gray(X2)

m, n = X1bw.shape
x1 = X1bw.reshape(m * n, 1)
x2 = X2bw.reshape(m * n, 1)
x1 = x1 - np.mean(x1)
x2 = x2 - np.mean(x2)

theta0 = 0.5 * np.arctan(-2 * np.sum(x1 * x2) / np.sum(x1 **
  2 - x2 ** 2))
Us = np.array([[np.cos(theta0), np.sin(theta0)],
  [-np.sin(theta0), np.cos(theta0)]])
```

This completes the first step in the SVD reconstruction process.

16.3.1.2 Step 2: Scaling of the principal components

The second step is to undo the scaling/compression that has been performed by the singular values along the principal component directions θ_0 and $\theta_0 - \pi/2$. The rescaling matrix is thus computed as follows:

```
sig1 = np.sum((x1 * np.cos(theta0) + x2 * np.sin(theta0)) ** 2)
sig2 = np.sum((x1 * np.cos(theta0 - np.pi/2) + x2 *
     np.sin(theta0 - np.pi/2)) ** 2)
Sigma = np.array([[1 / np.sqrt(sig1), 0], [0, 1 / np.sqrt(sig2)]])
```

Note that the variable called sigma above is actually the inverse of the diagonal matrix constructed from the SVD process. This completes the second step in the SVD process.

16.3.1.3 Step 3: Rotation to separability

The final rotation is aimed towards producing, as best as possible, a separable probability distribution. The analytic method used to do this is to minimize both the variance and kurtosis of the remaining distributions. The angle that accomplishes this task is computed analytically from either (16.2.14) or (16.2.15) and the associated rotation matrix $\mathbf{V}$ is given by (16.2.16). Before computing this final rotation, the image after steps 1 and 2 must be produced, i.e. we compute the $\bar{\mathbf{X}}_1$ and $\bar{\mathbf{X}}_2$ used in (16.2.14) and (16.2.15). The Python code used to produce the final rotation matrix is then

```
X1bar = Sigma[0, 0] * (Us[0, 0] * X1 + Us[0, 1] * X2)
X2bar = Sigma[1, 1] * (Us[1, 0] * X1 + Us[1, 1] * X2)

phi0 = 0.25 * np.arctan(-np.sum(2 * (X1bar ** 3) * X2bar -
   2 * X1bar * (X2bar ** 3)) /
                  np.sum(3 * (X1bar ** 2) * (X2bar ** 2) -
                     0.5 * (X1bar ** 4) -
                     0.5 * (X2bar ** 4)))

V = np.array([[np.cos(phi0), np.sin(phi0)], [-np.sin(phi0),
    np.cos(phi0)]])

S1bar = V[0, 0] * X1bar + V[0, 1] * X2bar
S2bar = V[1, 0] * X1bar + V[1, 1] * X2bar
```

This final rotation completes the three steps of computing the SVD matrix. However, the images needed to be rescaled back to the full image brilliance. Thus, the data points need to be made positive and scaled back to values ranging from 0 (dim) to 255 (bright). This final bit of code rescales and plots the separated images.

```
min1 = np.min(S1bar)
S1bar = S1bar - min1
max1 = np.max(S1bar)
S1bar = S1bar * (255 / max1)

min2 = np.min(S2bar)
S2bar = S2bar - min2
max2 = np.max(S2bar)
S2bar = S2bar * (255 / max2)
```

Note that the successive use of the **min/max** command is due to the fact that the colour images are $m \times n \times 3$ matrices, where the three levels give the colour mapping information.

16.3.2 Image separation from reflection

The algorithm above can be applied to the original problem presented of the **Judgement of Paris** by Flaxman. The original images are again illustrated in the top row of Fig. 16.10. The difference between the two figures was the application of a polarizer whose angle was rotated 90 degrees between shots. The bottom two figures show the extraction process that occurs for the two images. Ultimately, the technique applied here did not do such a great job. This is largely due to the fact that the reflected image was so significantly brighter than



Figure 16.10 The Judgement of Paris by Flaxman with a reflection of my window which overlooks Drumheller Fountain. In the top row are the two original photos taken with linear polarizer rotated 90 degrees between shots. The bottom row represents the separated images. The brightness of the reflected image greatly affects the quality of the image separation.

the original photographed image itself, thus creating a difficult image separation problem. Certainly, experience tells us that we also have this same difficulty: when it is very bright and sunny out, reflected images are much more difficult for us to see through to the true image.

16.4 **FastICA Algorithm**

As the previous sections show, the key to making ICA work is non-Gaussianity. In fact, most classical statistical theories assume random variables have a Gaussian distribution, which precludes the use of methods relating to ICA, as they explicitly require non-Gaussianity in order to determine the independent components. Non-Gaussianity in the previous chapters was measured by skewness and/or kurtosis. And in the image separation problem, kurtosis was the critical last step in determining the two independent components. A second important measure of non-Gaussianity is given by *negentropy*. Negentropy is an information-theoretic quantity that measures (differential) entropy. Entropy itself is the most basic concept of information theory. The entropy of a random variable approximately measures

the degree of information that the observation of the variable gives. Random variables have high entropy, while variables that have same structure or pattern have a lower entropy. It is directly related to the coding length of a variable, which is defined in information theory [136, 137].

The entropy of a discrete random variable Y is defined as

$$H(Y) = -\sum_j P(Y = a_j) \log P(Y = a_j) \tag{16.4.1}$$

where $P(\cdot)$ is the probability given the possible values a_j. This can be easily generalized to continuous variables, known as differential entropy, where the sum is replaced by an integral and the discrete probability is replaced by a probability density. Information theory guarantees that a Gaussian distributed random variable has the largest entropy among all random variables with equal variance.

To use entropy as a measure of non-Gaussianity, the negentropy is defined as follows:

$$J(\mathbf{y}) = H(\mathbf{y}_{\text{Gauss}}) - H(\mathbf{y}) \tag{16.4.2}$$

where $\mathbf{y}_{\text{Gauss}}$ is a Gaussian random variable with the same covariance as $\mathbf{y}$. If the variable $\mathbf{y}$ is Gaussian distributed, then the negentropy is zero and ICA methods fail. Negentropy is an optimal estimator of non-Gaussianity, making it useful for ICA and improvement over simply computing kurtosis. However, it is exceptionally difficult to estimate and compute in practice.

FastICA [138] is an algorithm that approximates the negentropy. Classical methods for approximating the negentropy make use of higher-order moments [139] in order to make the computation tractable. FastICA exploits an approximation based upon the maximum-entropy principle, whereby

$$J(\mathbf{y}) \approx \sum_{j=1}^{p} k_j \left[E\{G_j(\mathbf{y}\} - E\{G_j(\mathbf{v}\}\right]^2 \tag{16.4.3}$$

where the k_j are positive constants, $\mathbf{y}$ and $\mathbf{v}$ are variables with zero mean and unit variance, $\mathbf{v}$ is further Gaussian distributed, and G_j are some judiciously chosen nonquadratic functions. Advantageous choices of the G_j are what allow the FastICA algorithm to be an effective computational tool, with $G_1(\mathbf{y}) = (1/a_1) \log \cosh(a_1 \mathbf{y})$ $(a_1 \in [1, 2])$ and $G_2(\mathbf{y}) = -\exp(-\mathbf{y}^2/2)$ having proved to be good choices in practice [138]. This metric for non-Gaussianity can also be used in the simple computation of kurtosis in order to best separate the independent components.

Given the assumptions for computing the negentropy, data is pre-processed before computation of the ICA. Specifically, the theory assumes both mean-zero and unit variance data. These can be accomplished by centring the data and whitening the data, the former ensures that the mean is zero, while the latter ensures the variance to be unity. Other pre-processing tricks may be necessary based upon the application. For instance, band-pass filtering may be

advantageous for time signals before applying ICA, thus removing high-frequency content, which may undercut the ICA algorithm.

16.4.1 The FastICA Algorithm

The FastICA is an iterative scheme for finding the best representation of non-Gaussianity. FastICA starts with an initial vector $\mathbf{w}$, which acts as a projection from data $\mathbf{x}$ to a new coordinate $\mathbf{y} = \mathbf{w}^T\mathbf{x}$, where $J(\mathbf{w}^T\mathbf{x})$ is maximized, i.e. the non-Gaussianity or negentropy is largest in this projected direction.

The fixed-point iteration scheme is based upon a Newton iteration, which requires derivatives for the functions $G_1(\cdot)$ and $G_2(\cdot)$. Like activation function for neural networks, these functions are chosen for their explicit representation of derivatives, which are $g_1(\mathbf{y}) = \tanh a_1\mathbf{y}$ and $g_2(\mathbf{y}) = \mathbf{y}\exp(-\mathbf{y}^2/2)$. The algorithm proceeds as follows for a one-unit approximation $\mathbf{w}$

- Choose an initial (random) weight vector $\mathbf{w}$
- Let $\mathbf{w}^* = E\{\mathbf{x}g(\mathbf{w}^T\mathbf{x})\} - E\{\mathbf{x}g'(\mathbf{w}^T\mathbf{x})\}\mathbf{w}$
- Normalize $\mathbf{w} = \mathbf{w}^*/\|\mathbf{w}^*\|$
- Update $\mathbf{w}^*$ until converged

For computing multiple units of $\mathbf{w}$, i.e. weight vectors $\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_p$, where p is the number of independent components, the algorithm is adjusted to find one unit at a time. However, at each step, orthogonality between the $\mathbf{w}_k$ must be enforced through, for instance, a Gram–Schmidt procedure. Hyvärinen and Oja give an excellent review with detailed references for the FastICA scheme, both computational and theoretical [138].

16.4.2 Python Implementation of FastICA

FastICA has been implemented through the scikit-learn package in Python. Thus, perhaps the ICA algorithm is available for broad usage. In the previous section, images where mixed from two pictures of Sicily. The flattened and mixed vectors were $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. The following code sends the two measurements into the FastICA algorithm, which then produces an estimate of the independent signals and the mixing matrix.

```
from sklearn.decomposition import PCA, FastICA

X = np.hstack((x1,x2))

ica = FastICA(n_components=2, whiten="arbitrary-variance")
S_ = ica.fit_transform(X)  # Reconstruct signals
A_ = ica.mixing_  # Get estimated mixing matrix
```

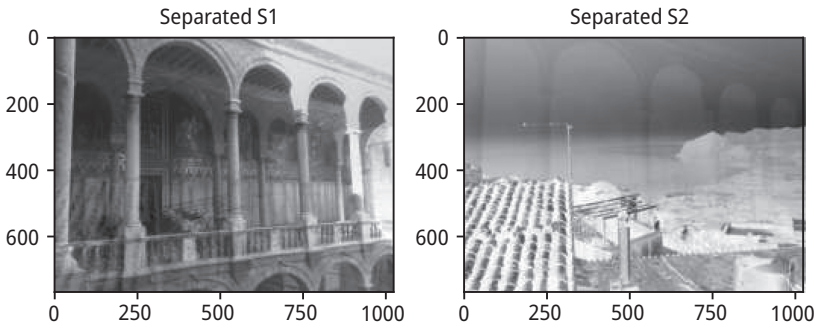


Figure 16.11 Separation of images. The reconstructed images from the image separation process and approximation using FastICA. FastICA does better than the manual tuning used for Fig. 16.9.

Figure 16.11 shows the results of the FastICA algorithm in separating the images. FastICA also gives an approximation to the mixing matrix, which was used for separating the two images.

17 Unsupervised Machine Learning

17.1 Data Mining, Feature Spaces, and Clustering 484

17.2 Unsupervised Clustering Algorithms 488

17.2.1 The k-means algorithm 489

17.2.2 Hierarchical cluster algorithm and dendrograms 490

17.2.3 Mixture models: Gaussian Mixture Models (GMMs) 492

Data mining is a critical task for the characterization of the most salient features of a given data set. More than that, data mining is the first step for producing diagnostics for prediction and forecasting. Unsupervised learning is essentially equivalent to data mining since it assumes that the data is unlabelled and has unknown features. Thus, unsupervised learning techniques aim to not only provide labels, or groupings, for the data set considered, but also to define a *feature space*, which allows the algorithm to automatically extract the dominant features enabling the clustering and classification of data. Unsupervised learning is much more difficult than supervised learning due to the lack of prescribed labels and expert knowledge on the feature space. In this chapter, the three most common algorithms for unsupervised learning will be considered. They are simple and largely effective algorithmic ways to automate the extraction of information and features from unknown data.

17.1 Data Mining, Feature Spaces, and Clustering

To learn about data, it is first necessary to extract the most meaningful features of the data. A well-constructed *feature space* is of exceptional value in terms of allowing one to cluster and classify the data being considered. For unsupervised learning, we once again consider its mathematical framing by considering extraction of a limited set of representative data from a given domain

$$\mathcal{D}' \subset \mathbb{R}^n \tag{17.1.1}$$

where $\mathcal{D}'$ is an open bounded set of dimension n. The data itself is a subset of all possible data so that

$$\mathcal{D}' \subset \mathcal{D}\,. \tag{17.1.2}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0018

where $\mathcal{D}$ is all possible data. When introduced in Ch. 6, the data was represented by data points $\mathbf{x}_j \in \mathbb{R}^n$, which can be used to construct the input data matrix $\mathbf{X}$. Although not discussed then, the $\mathbf{x}_j$ used for generating classification labels is called the feature space. Thus, the output as prescribed in (6.1.6) is given by the labels $\mathbf{y}_j$ for each point where $j = 1, 2, \ldots, m$. The label vectors for this feature space can be used to construct the output matrix $\mathbf{Y}$. Thus, the input to the model is the data $\mathbf{X}$, and the output to the model is a label encoded in $\mathbf{Y}$. As noted previously, labels for the data are diverse and include numeric values, including integers and test strings.

The goal in data mining is partly to construct a feature space $\mathbf{x}_j$, which allows for robust and accurate classification of the data. Of course, for unsupervised learning, there are no labels. Thus, the input is designed to elicit an output that is a classification label. Mathematically, the unsupervised task is represented as follows:

Input

$$\text{data} \quad \{\mathbf{x}_j \in \mathbb{R}^n,\ j \in Z := \{1, 2, \ldots, m\}\} \tag{17.1.3a}$$

Output

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z\}\,. \tag{17.1.3b}$$

Thus, the unsupervised learning goal is not only aimed at producing labels $\mathbf{y}_j$ for all the data, but also a feature space allowing for good clustering of the data. Preferably, this feature space spanned by $\mathbf{X}$ would be interpretable and allow for better understanding of the data.

One of the earliest and practical data sets used for learning algorithms was the MNIST (Modified National Institute of Standards and Technology) data set [140, 141]. This data set features hand-written numerical digits from 0 to 9 as exemplified by Fig. 17.1. Supervised learning, whose goal was to accurately cluster the digits with full knowledge that they represented digits from 0 to 9. The MNIST data set was of exceptional value to the United States Postal Service (USPS) in the development of automated tools for reading zip codes and automated sorting of mail.

In Ch. 18, we will endow our machine learning algorithms with knowledge and labels of the MNIST data set. But for this chapter, the consideration is aimed at the automated extraction, or understanding, of the digits from 0 to 9. Thus, one can ask the following question: can the computer learn that there are 10 distinct digits used in our numeric system by simply looking at a large collection of hand-written samples of numbers? It is an unsupervised task since no knowledge of the digits is given to the algorithm. The data itself is a 28 by 28 pixel array, which is a greyscale representation of a numerical digit. This gives an initial feature space of dimension 784, which represents all the pixels of the digit. Instead of working directly with the pixel space, we take the data and look at its principle component space (PCA), which is illustrated in Fig. 17.2. In fact, we collapse the data into the dominant correlations in the data using PCA and treat this as the feature space in which unsupervised learning occurs. The following code imports the MNIST data and extracts the data to arrays for processing.

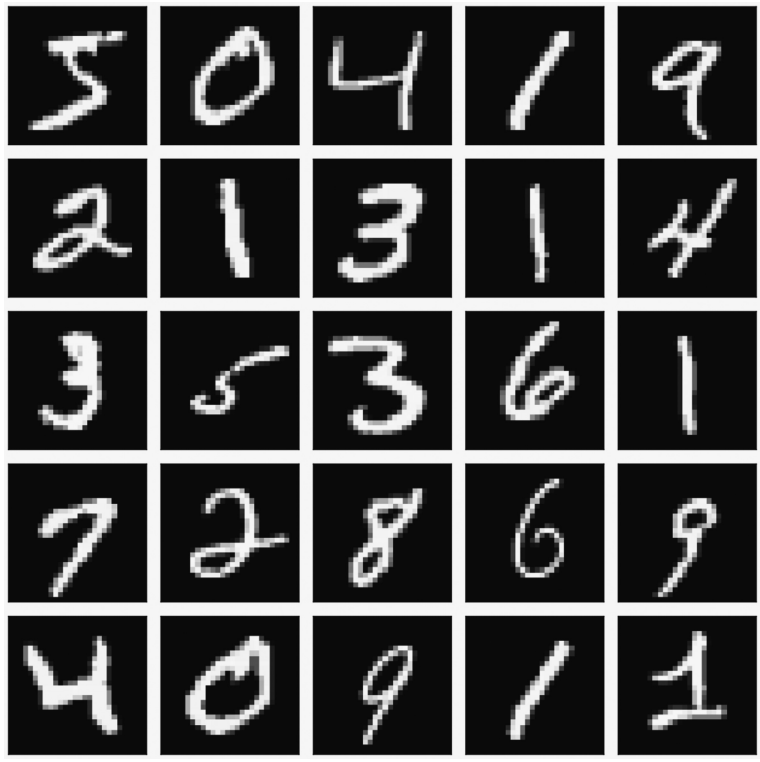


Figure 17.1 Example of hand-written digits in the MNIST (Modified National Institute of Standards and Technology) data set [140, 141]. The data represents digits from 0 to 9 in 28 by 28 pixel images. MNIST was an early data set used for both unsupervised learning, whose goal was to learn and classify the 10 digits.

```
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA

# Load the MNIST data
mnist = fetch_openml('mnist_784', parser='auto')
X = mnist.data / 255.0  # Scale the data to [0, 1]
y = mnist.target

# Apply PCA to reduce the dimensionality of the data
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)
X_pca = np.array(X_pca)
y_pca = np.array(y)
```

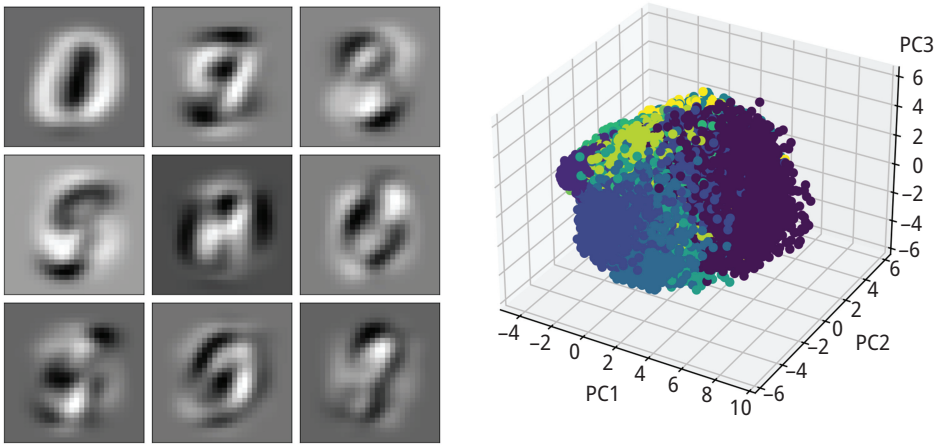


Figure 17.2 The top left panel shows the first nine PCA (SVD) modes of the MNIST data set. In what follows, we use the first three PCA modes to learn a classifier for recognition of hand-written digits. The digit clusters for all 70,000 hand-written digits projected into the three PCA mode space is shown in the right panel. One can see that even in this PCA feature space, there emerges distinct clusters for the various digits. Each distinct colour represents a different digit.

This code creates a data matrix $\mathbf{X}_{pca}$ that is now 70,000 by 3. Thus, there are 70,000 example digits, approximately 7000 of each digit, which are all collapsed into the three leading PCA modes. The goal of unsupervised learning is to build a classifier that can learn the 10 numeric digits in order to accurately classify new digits not used in the training set. Figure 17.2 shows how the data can be clustered in the PCA space. This is just one example of a feature space that can be constructed.

To more easily visualize the task of unsupervised clustering, we consider a subset of the MNIST data set. Specifically, consider the hand-written digits associated with a 1 or a 4 as shown in Fig. 17.3. One thousand example images of these two digits can be projected into the PCA space as shown in Fig. 17.4. On the left panel of this figure is how the data would be initially visualized without labels. Even without labels, there appears to be two distinct clusters of the data. By colouring them by their label of either 1 or 4, the digits are indeed seen to be well separated. This figure illustrates the main concept in unsupervised learning: label the different clusters in some appropriate manner in order to extract information about these clusters in a given feature space. A good feature space should allow for nicely separated clusters. A poor feature space does not separate the data well into distinct clusters.

Thus far, we have only considered the construction of the feature space, which was done using a PCA decomposition. What remains is to automate the building of clusters and classification of data in an algorithmic way. The combination of feature space and automated separation of the data constitutes the key components of data mining. What follows in this chapter are three of the classic data mining techniques for unsupervised learning: k-means clustering, hierarchical clustering and Gaussian mixture models. Each is defined by an algorithm that searches for the best similarity and separation of the data in a given feature space.

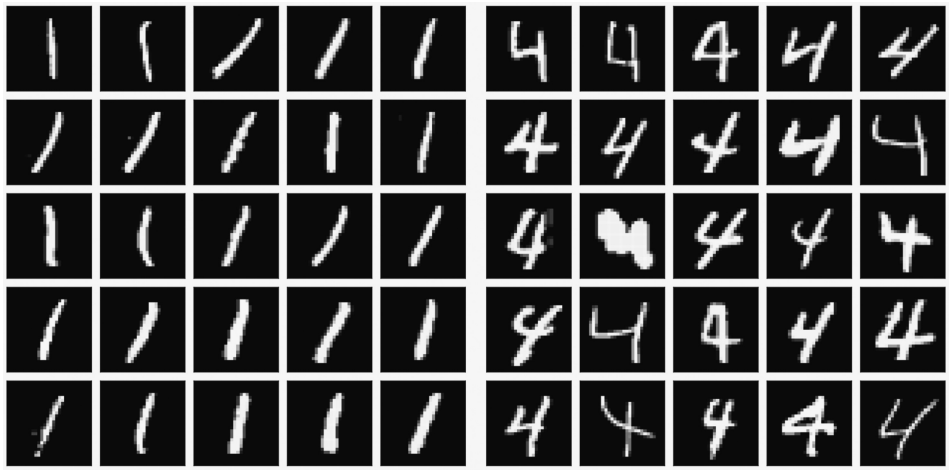


Figure 17.3 Examples of the hand-written digits 1 and 4 represented as 28 by 28 pixel arrays. Approximately, 7000 examples of each are given in the MNIST data set. The leading components of PCA space allows for an effective clustering between these two digits as shown in Fig. 17.4.

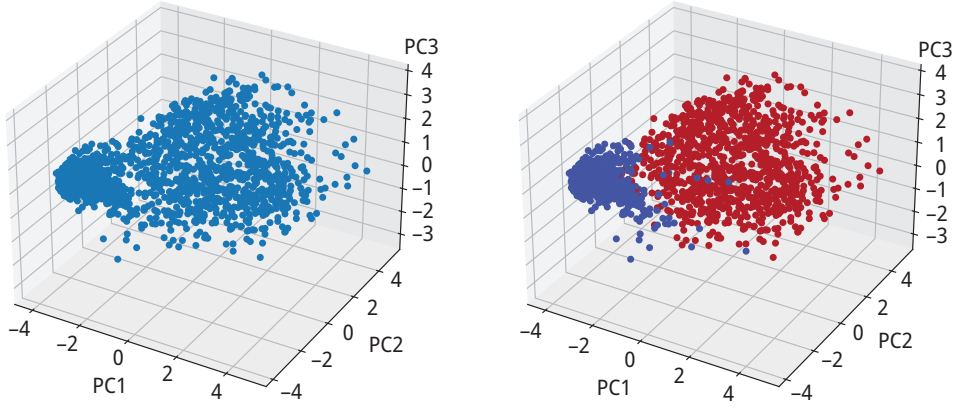


Figure 17.4 Clustering in PCA space of the digits 1 and 4. The left panel represents the data that is initially given to the user for clustering and classification. No labels are provided. A good unsupervised algorithm and feature space would be able to produce labels for the data, which are meaningful. The right panel shows the data coloured by whether the digits are 1 and 4. The two digits are well separated in the PCA feature space used here.

17.2 **Unsupervised Clustering Algorithms**

Without training labels, extracting meaningful patterns and features of data is generally quite challenging. Simple algorithms are often promoted as they can be applied on large data sets with a minimum of assumptions. The first assumption to be made generically is on the number of clusters to be extracted. This is a user-defined hyper-parameter that can be instrumental in determining a good extraction of clusters from data. In the example shown

in the previous section with MNIST, we actually know there are 10 distinct clusters associated with the digits. However, in practice this would be unknown, making it significantly more difficult.

To begin an unsupervised learning algorithm, the number of partitions k must be initially specified. Alternatively, one can posit some criteria for what makes a good number of clusters based upon a prescribed objective function. One should be aware that there are two natural asymptotic limits: $k = 1$, whereby all the data is one cluster, and $k = m$, whereby every data point is its own cluster. Of course, these two limits tell you nothing. What is desired is a *parsimonious* number of clusters that balances a minimal number of clusters with good performance in classification. Once a number of partitions, or clusters k, is selected, the algorithm can proceed.

17.2.1 The k-means algorithm

The k-means algorithm, which was published in 1982, was actually first proposed by Stuart Lloyd in 1957 [142]. It is a very simple algorithm in practice: once the number of partitions k is specified, one simply assigns membership of a given data point to the nearest cluster centre. This step is repeated once the cluster centre positions are updated by considering the position of their *centre-of-mass*. Figure 17.5(a) depicts the k-means algorithm graphically. More precisely, the algorithm proceeds as follows: (i) given initial values for k distinct means, compute the distance of each observation $\mathbf{x}_j$ to each of the k means. (ii) Label each observation as belonging to the nearest mean. (iii) Once labelling is completed, find the *centre-of-mass* (mean) for each group of labelled points. These new means are then used to start back at step (i) in the algorithm. From the viewpoint of optimization, the k-means algorithm aims to minimize the following objective:

$$\text{argmin}_{\boldsymbol{\mu}_j} \sum_{j=1}^{k} \sum_{\mathbf{x}_j \in \mathcal{D}'_j} \|\mathbf{x}_j - \boldsymbol{\mu}_j\|^2 \tag{17.2.1}$$

where $\boldsymbol{\mu}_j$ denotes the mean of the jth cluster and $\mathcal{D}'_j$ denotes the subdomain of data associated with that cluster. This minimizes the within-cluster sum of squares. In general, solving the optimization problem as stated is NP-hard, making it computationally intractable. However, there a number of heuristic algorithms that provide good performance despite not having a guarantee that they will converge to the globally optimal solution.

The k-means algorithm is easy to implement within Python. The following code takes a data set $\mathbf{X}$ whose columns are the observations $\mathbf{x}_j$, where $j = 1, 2, \cdots, m$. Since it is applied to MNIST data, 10 clusters are selected.

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=10)
kmeans.fit(X)
labels = kmeans.labels_
cluster_centers = kmeans.cluster_centers_
```

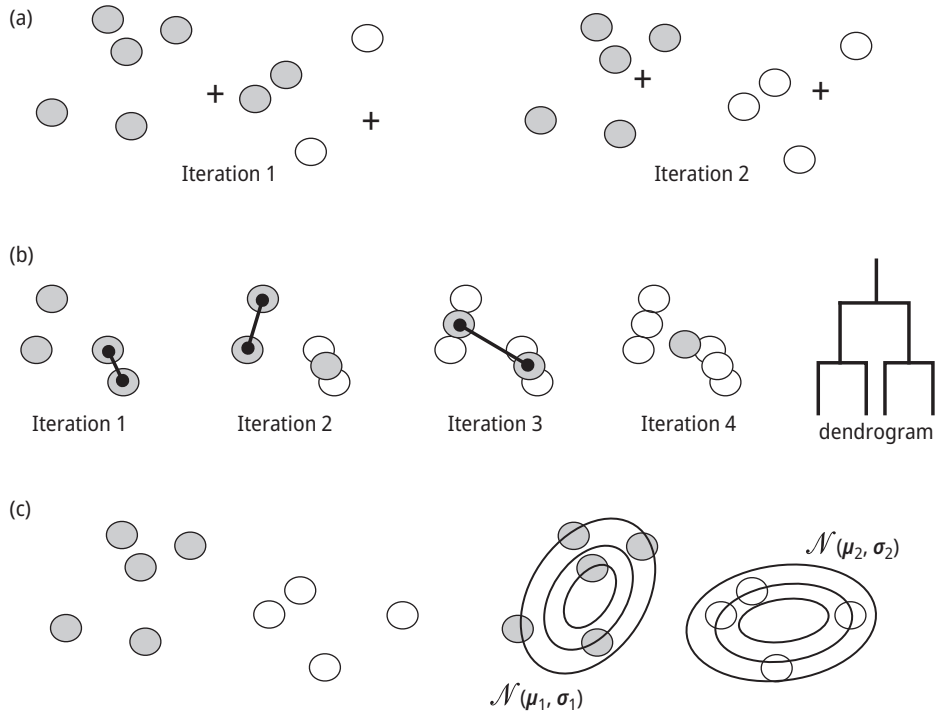


Figure 17.5 Visualization of the (a) k-means algorithm, (b) hierarchical clustering, and (c) Gaussian mixture models. In the k-means algorithm, the initial cluster centres ($+$) include all data points in closest proximity. The new cluster centres are found from centre-of-mass of each membership. The hierarchical clustering agglomerates points by which points are closest together in a given metric until only a single point remains. And finally, GMMs simply give the best fit to the data for a prescribed number of Gaussians.

The labels for the data and their cluster centres are both extracted from the fitting process. New data can be classified by simply seeing which cluster centre it is closest to. Applying the algorithm to the MNIST data set is shown in Fig. 17.6. Only 300 data points are considered in order to aid in visualization of the k-means clustering algorithm.

17.2.2 Hierarchical cluster algorithm and dendrograms

A second commonly used unsupervised algorithm for automating clustering is known as hierarchical clustering. As with k-means, the algorithm proceeds in a straight-forward fashion by considering the distance between each data pair $\mathbf{x}_j$ and $\mathbf{x}_k$ (although k-means considers the distance from each point to the current cluster centre). Although we typically use a Euclidean distance, there are a number of important distance metrics one might consider for different types of data. Some typical distances are given as follows:

$$\text{Euclidean distance } \|\mathbf{x}_j - \mathbf{x}_k\|_2 \tag{17.2.2a}$$

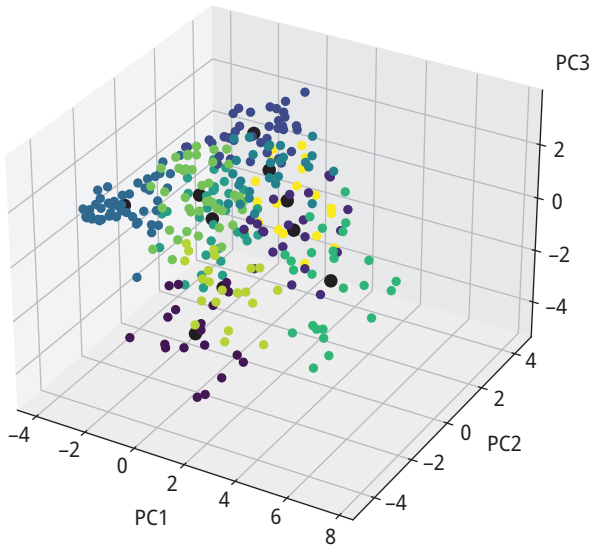


Figure 17.6 Clustering of first 300 digits of the MNIST data set in PCA space. The k-means algorithm produces 10 distinct cluster centres, which are aimed at producing representations in PCA space of the digits from 0 to 9. The cluster centres found by k-means are the large black dots.

$$\text{Squared Euclidean distance} \quad \|\mathbf{x}_j - \mathbf{x}_k\|_2^2 \tag{17.2.2b}$$

$$\text{Manhattan distance} \quad \|\mathbf{x}_j - \mathbf{x}_k\|_1 \tag{17.2.2c}$$

$$\text{Maximum distance} \quad \|\mathbf{x}_j - \mathbf{x}_k\|_\infty \tag{17.2.2d}$$

$$\text{Mahalanobis distance} \quad \sqrt{(\mathbf{x}_j - \mathbf{x}_k)^T \mathbf{C}^{-1} (\mathbf{x}_j - \mathbf{x}_k)} \tag{17.2.2e}$$

where $\mathbf{C}^{-1}$ is the covariance matrix. As already illustrated in Chapter 15, the choice of norm can make a tremendous difference for exposing patterns in the data that can be exploited for clustering and classification.

Clustering data is visualized through a *dendrogram* by applying an agglomerative method, whereby initially each data point $\mathbf{x}_j$ is its own cluster. The data is merged in pairs, the closest two points merged first, as one creates a hierarchy of clusters. Figure 17.5(b) depicts the k-means algorithm graphically. The merging of data eventually stops once all the data has been merged into a single über cluster. This is the bottom-up approach in hierarchical clustering. One can instead do divisive clustering, whereby all the observations $\mathbf{x}_j$ are initially part of a single giant cluster. The data is then recursively split into smaller and smaller clusters. The splitting continues until the algorithm stops according to a user-specified objective. The divisive method can split the data until each data point is its own node. In general, the merging and splitting of data is accomplished with a heuristic, greedy algorithm, which is easy to execute computationally. The following code produces two dendrogram representations: one which shows the entire dendrogram is constructed from:

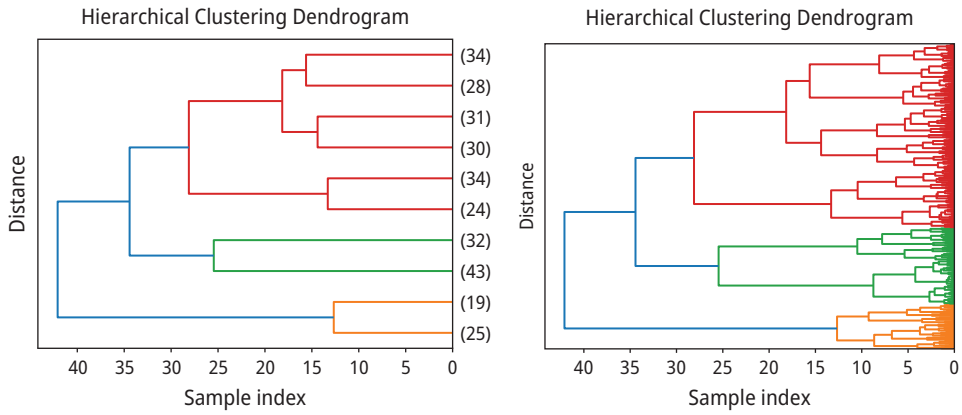


Figure 17.7 Dendrograms produced from hierarchical clustering algorithms. The left panel shows the clustering into 10 distinct clusters with the total number of digits assigned to each cluster. This gives a cluster-centric view of the membership of the data. The right panel shows the clustering of all 300 data points along with their distances relative to each other. Visualizing even 300 data points is difficult with a dendrogram. Thus, the left panel can often be a better visualization tool.

```
from scipy.cluster.hierarchy import dendrogram, linkage, cut_tree

Z = linkage(X, 'ward')
dendrogram(Z, labels=y, orientation='left', truncate_mode='lastp', p=10)
dendrogram(Z, labels=y, orientation='left')
```

The dendrogram algorithm is shown in Fig. 17.7 for the first 300 MNIST data points. The algorithm is as follows: (i) the distance between all m data points $\mathbf{x}_j$ is computed (the figure illustrates the use of a Euclidean distance), (ii) the closest two data points are merged into a single new data point midway between their original locations, and (iii) the calculation is repeated with the new $m-1$ points. The algorithm continues until the data has been hierarchically merged into a single data point. One can either view the membership in aggregate for the 10 class clustering, as in the left panel of Fig. 17.7, or the clustering for all 300 data points as seen in the right panel.

17.2.3 Mixture models: Gaussian Mixture Models (GMMs)

The third most common unsupervised learning method is *Gaussian mixture models* (GMMs). This model assumes that the data is drawn from a distinct number of Gaussian distributions. Of course, the distributions do not have to be Gaussian, thus this modelling framework falls under the broader aegis of *finite mixture models*. As with k-means and hierarchical clustering,

it is assumed that k processes are combined to form the measurements and data observations $\mathbf{x}_j$. Thus, there are k mixtures of individual statistical distributions that best fit the data. GMMs assume that each mixture model has a Gaussian distribution, which allow for a complete characterization by two parameters: the mean and the variance. Figure 17.5(c) depicts the k-means algorithm graphically.

GMMs are evaluated using maximum-likelihood and the underlying *Expectation-Maximization* (EM) algorithm of Dempster, Laird, and Rubin [143]. The iterative structure of the algorithm finds a local maximum-likelihood, which estimates the true parameters, in this case mean and variance, that cannot be directly solved for. The algorithm has an iterative construction that recursively estimates the best parameters possible from an initial guess. The convergence of such iteration schemes is always important to consider, something which cross-validation can help to evaluate if a bad initial guess was chosen.

Mixture models assume the probability density function (PDF) for observations of data $\mathbf{x}_j$ is a weighted linear sum of a set of unknown distributions

$$f(\mathbf{x}_j, \boldsymbol{\Theta}) = \sum_{p=1}^{k} \alpha_p f_p(\mathbf{x}_j, \boldsymbol{\Theta}_p) \tag{17.2.3}$$

where $f(\cdot)$ is the measured PDF, $f_p(\cdot)$ is the PDF of the mixture j, and k is the total number of mixtures. Each of the PDFs $f_j(\cdot)$ is weighted by α_p ($\alpha_1 + \alpha_2 + \cdots + \alpha_k = 1$) and parametrized by an unknown vector of parameters $\boldsymbol{\Theta}_p$. To state the objective of mixture models more precisely then: *Given the observed PDF* $f(\mathbf{x}_j, \boldsymbol{\Theta})$*, estimate the mixture weights* α_p *and the parameters of the distribution* $\boldsymbol{\Theta}_p$. Note that $\boldsymbol{\Theta}$ is a vector containing all the parameters $\boldsymbol{\Theta}_p$. Making this task somewhat easier is the fact that we assume the form of the PDF distribution $f_p(\cdot)$. For GMMs, the parameters in the vector $\boldsymbol{\Theta}_p$ are known to include only two variables: the mean μ_p and variance σ_p, and $f_p(\mathbf{x}_j, \boldsymbol{\Theta}_p) = \mathcal{N}_p(\mathbf{x}_j, \mu_p, \sigma_p)$, which gives a more tractable framework since there are now a limited set of parameters. Thus, once one assumes a number of mixtures k, then the task is to determine α_p along with μ_p and σ_p.

An estimate of the parameter vector $\boldsymbol{\Theta}$ can be computed using the *maximum likelihood estimate* (MLE) of Fisher. The MLE computes the value of $\boldsymbol{\Theta}$ from the roots of $\partial L(\boldsymbol{\Theta})/\partial \boldsymbol{\Theta} = 0$, where the log-likelihood function L is

$$L(\boldsymbol{\Theta}) = \sum_{j=1}^{n} \log f(\mathbf{x}_j | \boldsymbol{\Theta}) \tag{17.2.4}$$

and the sum is over all the n data vectors $\mathbf{x}_j$. The solution to this optimization problem, i.e. when the derivative is zero, produces a local maximizer. This maximizer can be computed using the EM algorithm since derivatives cannot be explicitly computed without an analytic form.

The EM algorithm starts by assuming an initial estimate (guess) of the parameter vector $\boldsymbol{\Theta}$. This estimate can be used to estimate

$$\tau_p(\mathbf{x}_j, \boldsymbol{\Theta}) = \frac{\alpha_p f_p(\mathbf{x}_j, \boldsymbol{\Theta}_p)}{f(\mathbf{x}_j, \boldsymbol{\Theta})} \tag{17.2.5}$$

which is the posterior probability of component membership of $\mathbf{x}_j$ in the pth distribution. In other words, does $\mathbf{x}_j$ belong to the pth mixture? The E-step of the EM algorithm uses this posterior to compute memberships. For GMM, the algorithm proceeds as follows: Given an initial parametrization of $\boldsymbol{\Theta}$ and α_p, compute

$$\tau_p^{(k)}(\mathbf{x}_j) = \frac{\alpha_p^{(k)} \mathcal{N}_p(\mathbf{x}_j, \mu_p^{(k)}, \sigma_p^{(k)})}{\mathcal{N}(\mathbf{x}_j, \boldsymbol{\Theta}^{(k)})}. \tag{17.2.6}$$

With an estimated posterior probability, the M-step of the algorithm then updates the parameters and mixture weights

$$\alpha_p^{(k+1)} = \frac{1}{n} \sum_{j=1}^{n} \tau_p^{(k)}(\mathbf{x}_j) \tag{17.2.7a}$$

$$\mu_p^{(k+1)} = \frac{\sum_{j=1}^{n} \mathbf{x}_j \tau_p^{(k)}(\mathbf{x}_j)}{\sum_{j=1}^{n} \tau_p^{(k)}(\mathbf{x}_j)} \tag{17.2.7b}$$

$$\Sigma_p^{(k+1)} = \frac{\sum_{j=1}^{n} \tau_p^{(k)}(\mathbf{x}_j) \left(\mathbf{x}_j - \mu_p^{(k+1)}\right) \left(\mathbf{x}_j - \mu_p^{(k+1)}\right)^T}{\sum_{j=1}^{n} \tau_p^{(k)}(\mathbf{x}_j)} \tag{17.2.7c}$$

where the matrix $\Sigma_p^{(k+1)}$ is the covariance matrix containing the variance parameters. The E- and M-steps are alternated until convergence within a specified tolerance. Recall that to initialize the algorithm, the number of mixture models k must be specified and initial parametrization (guesses) of the distributions given. This is similar to the k-means algorithm where the number of clusters k is prescribed and an initial guess for the cluster centres is specified.

The algorithm implementation is as follows in Python:

```
from sklearn.mixture import GaussianMixture

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2)
gmm = GaussianMixture(n_components=10, covariance_type='diag')
gmm.fit(X_train)
y_pred = gmm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

means = gmm.means_
variances = gmm.covariances_
```

This produces the parameters of each of the Gaussian distributions. Specifically, their means and variances, which is all that is needed for the GMMs.

Supervised Machine Learning

18

18.1 Supervised Learning: Recognizing Dogs and Cats 496
18.1.1 Wavelet analysis of images and the SVD and LDA 497
18.1.2 Wavelet decomposition 497
18.2 The SVD and Linear Discrimination Analysis 501
18.2.1 The SVD decomposition 502
18.2.2 Linear discrimination analysis 504
18.3 Classification Trees, Support Vector Machines, and Neural Networks 508
18.3.1 Classification Trees 508
18.3.2 Support Vector Machines 510
18.3.3 Neural Networks 512

In the last chapter, we discussed feature spaces and data mining for unsupervised learning. The goal was to learn and extract information from the data in order to better characterize patterns embedded in it. In this chapter, we instead turn our attention to supervised learning. The data is now provided with labels, which already removes many of the issues associated with data mining. Specifically, we know what the data is. One of the most intriguing uses of supervised learning over the past two decades concerns the ideas of image recognition. In particular, mathematically plausible methods are being rapidly developed to mimic real biological processes towards automating computers to recognize people, animals, and places. Indeed, there are many applications, many of them available on smart phones, that can perform sophisticated algorithms for doing the following: tag all images in a photo library with a specific person, and automatically identify the location and name of a set of mountain peaks given a photo of the mountains. These image recognition methods are firmly rooted in the basic mathematical techniques to be described in this chapter. The pace of this technology and its potential for scientific impact is remarkable. The methods here will bring together statistics, time–frequency analysis, and the SVD.

Taken as a whole, the dimensionality reduction and statistical methods used in supervised and unsupervised learning are simple examples of so-called *machine learning*, otherwise known as *statistical learning*. More broadly, such data classification schemes fall under pattern theory and are becoming a dominant paradigm in computer science for handling *big data*. In this example, a *supervised learning* technique is implemented in which a prescribed set of data is known and classified beforehand. Alternatively, one could modify the code

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.
© J. Nathan Kutz (2026). DOI: 10.1093/oso/9780198929093.003.0019

to do *unsupervised learning* so that the code itself learns to classify and group the data in an appropriate fashion. We will not dwell on these issues here, but will simply present an intuitive example of the machine learning architecture.

18.1 Supervised Learning: Recognizing Dogs and Cats

Recall from Ch. 6 that the supervised learning classification task as a function of input and output can be stated as follows:

$$\textbf{Input}$$

$$\text{data} \quad \{\mathbf{x}_j \in \mathbb{R}^n,\ j \in Z := \{1, 2, \ldots, m\}\} \tag{18.1.1a}$$

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z' \subset Z\} \tag{18.1.1b}$$

$$\textbf{Output}$$

$$\text{labels} \quad \{\mathbf{y}_j \in \{\pm 1\},\ j \in Z\} \tag{18.1.1c}$$

where a subset of the data is labelled and the missing labels are provided for the remaining data. For supervised learning, all the labels are known in order to build the classifier model $f_{\boldsymbol{\theta}}(\cdot)$ on $\mathcal{D}'$. The classifier is then applied to all data in $\mathcal{D}$. Thus, unlike the unsupervised learning scenario in which no knowledge of the data (or labels) are provided, supervised learning already has critical information about the nature of the data encoded in the label. In the example that follows, a supervised algorithm will be implemented for a computer vision task. The advantage of this application is that one can directly see how the supervised learning algorithm functions.

A basic and plausible methodology will be formulated in the context of image recognition. Specifically, we will generate a mathematically plausible mechanism by which we can discriminate between dogs and cats with labelled training data. Before embarking on this pursuit, however, it is useful to consider our own perception and image recognition capabilities. To some extent, the art world has been far ahead of the mathematicians and scientists on this, often asking the question about how a figure can be minimally represented. For instance, consider the Picasso-inspired image represented in Fig. 18.1. With very simple lines, Picasso was able to represent a dog. This example is particular striking given that not only do we recognize a dog, but we recognize the specific dog represented: a dachshund. It is clear from these examples that the concept of edge detection must play a critical role in the image recognition problem. This highlights the critical feature space required for assessment. The question to ask is this: could a computer be trained to recognize the figures and animals represented by Picasso in these works?

Picasso certainly pushed much further in the representation of figures and images. Indeed, many of his paintings challenge our ability to perceive the figures within them. But the fact remains, we often do perceive the figures represented even if they are remarkably abstract and distorted from our normal perception of what a human figure should look like, and yet there remains clear indications to our senses of the figures within the abstractions. This highlights the amazingly robust imaging system available to humans.

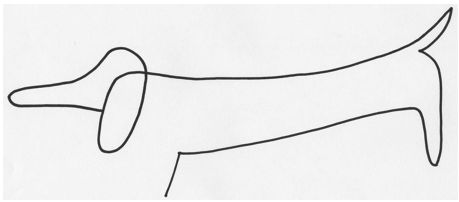


Figure 18.1 Author's sketch of a dog inspired by Picasso's own sketch of a dog. Note the identification of the image occurs simply from drawing the edges of the animal.

18.1.1 Wavelet analysis of images and the SVD and LDA

As discussed previously in the time–frequency analysis, wavelets represent an ideal way to represent multiscale information. As such, wavelets will be used here to represent images of dogs and cats. Specifically, wavelets are tremendously efficient in detecting and highlighting edges in images. In the image detection of cats versus dogs, the edge detection will be the dominant aspect of the discrimination process.

The mathematical objective is to develop an algorithm by which we can train the computer to distinguish between dogs and cats. The algorithm will consist of the following key steps:

- **Step 1** Decompose images of dogs and cats into wavelet basis functions. The motivation is simply to provide an effective method for doing edge detection. The training sets for dogs and cats will be 80 images each.
- **Step 2** From the wavelet-expanded images, find the principal components associated with the dogs and cats, respectively.
- **Step 3** Design a statistical decision threshold for discrimination between the dogs and cats. The method to be used here will be a *linear discrimination analysis* (LDA).
- **Step 4** Test the algorithm efficiency and accuracy by running through its own withheld pictures of cats and dogs.

The focus of the remainder of this section will simply be on step 1 of the algorithm and the application of a wavelet analysis.

18.1.2 Wavelet decomposition

To begin the analysis, a sample of the dog pictures is presented. The file **dogData.mat** contains a matrix $\mathbf{dog}$ which is a 4096×80 matrix. The 4096 data points correspond to a 64×64 resolution image of a dog, while the 80 columns are 80 different dogs for the training set of dogs. A similar file exists for the cat data called **catData.mat**. The following command lines load the data and plot the first nine dog faces.

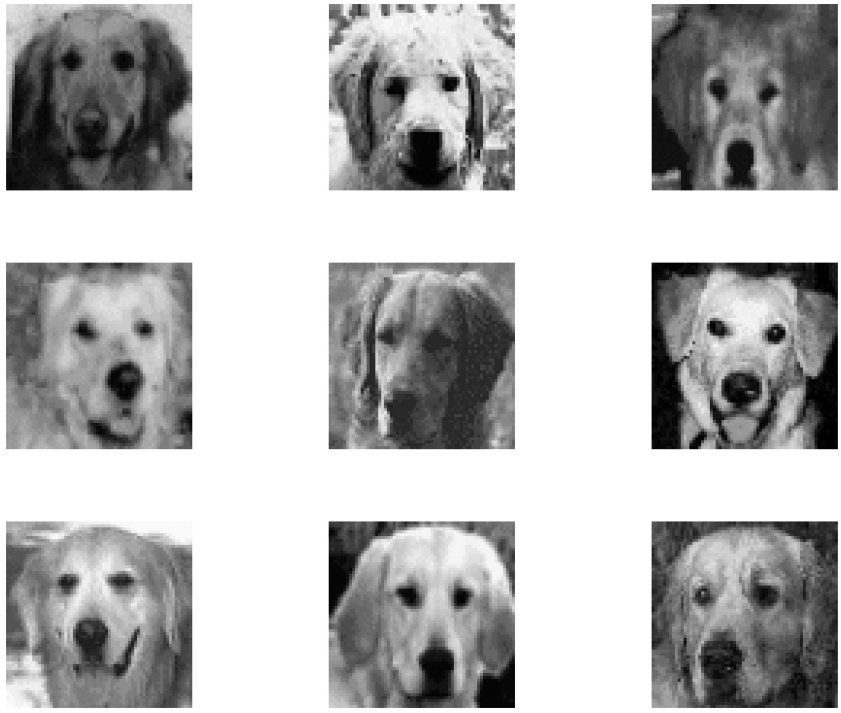


Figure 18.2 A sampling of nine dog faces to be used in the training set for the dog versus cat recognition algorithm.

```
dog_data = loadmat('dogData.mat')
dog = dog_data['dog']

for j in range(9):
    plt.subplot(3, 3, j + 1)
    dog1 = np.reshape(dog[:, j], (64, 64))
    plt.imshow(dog1, cmap='gray')
```

Figure 18.2 demonstrates nine typical dog faces that are used in training the algorithm to recognize dogs versus cats. Note that high-resolution images will not be required to perform this task. Indeed, a fairly low resolution can accomplish the task quite nicely.

These images are now decomposed into a wavelet basis using the discrete wavelet transform in Python. In particular, the **dwt2** command performs a single-level discrete 2D wavelet transform, or decomposition, with respect to either a particular wavelet or particular wavelet filters that you specify. The command

```
X = np.double(np.reshape(dog[:, 5], (64, 64)))
cA, (cH, cV, cD) = pywt.dwt2(X, 'haar')
```

computes the approximation coefficient matrix **cA** and details coefficient matrices **cH**, **cV**, **cD**, obtained by a wavelet decomposition of the input image matrix **X**. The input **'haar'** is a string containing the wavelet name. Thus, the large-scale features are in **cA** and the fine-scale features in the horizontal, vertical, and diagonal directions are in **cH**, **cV**, **cD**, respectively.

The Haar wavelet decomposition produces four 32×32 matrices. Figure 18.3 shows the wavelet decomposition matrices for the sixth dog in the data set. Part of the reason the images are so dark in Fig. 18.3 is that the images have not been rescaled to the appropriate pseudo-colour scaling used by the image commands. The following lines of code not only rescale the images but also combine the edge detection wavelets in the horizontal and vertical direction into a single matrix.

```
nbcol = plt.cm.gray.N
cod_cH1 = pywt.wavedec2(cH, 'haar', level=1)
cod_cV1 = pywt.wavedec2(cV, 'haar', level=1)
cod_edge = np.abs(cod_cH1[0]) + np.abs(cod_cV1[0])
```

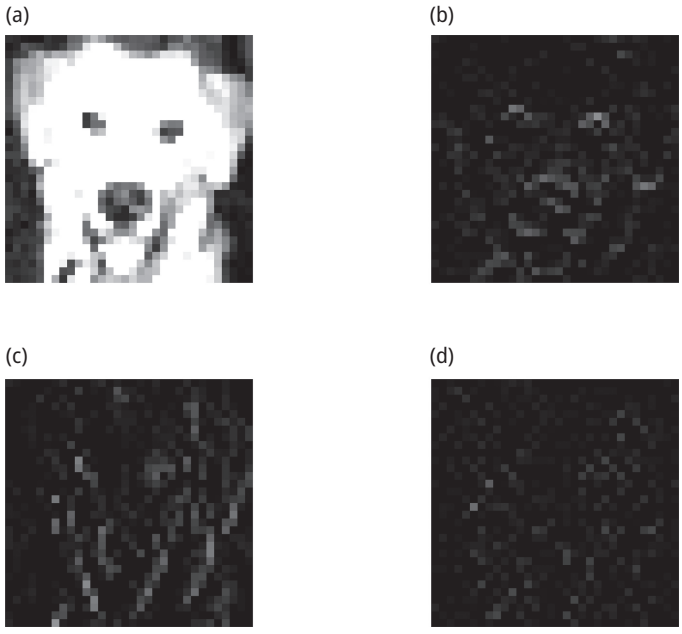


Figure 18.3 Wavelet decomposition into the matrices (a) **cA** (large-scale structures), (b) **cH** (fine scales in the horizontal direction), (c) **cV** (fine scales in the vertical direction), and (d) **cD** (fine scales in the diagonal direction).

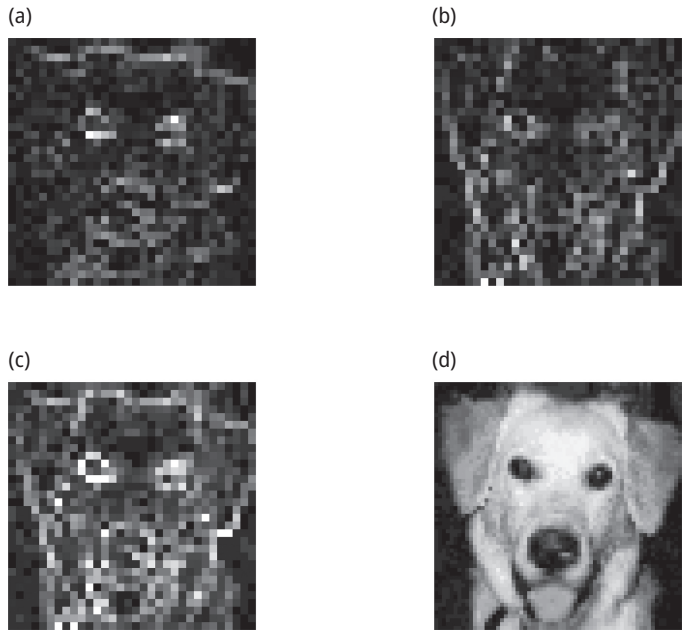


Figure 18.4 This figure represents the (a) horizontal and (b) vertical representations of the wavelet transform in the appropriately scaled image scale. The overall wavelet transformation process is represented in (c), where the edge detection in the horizontal and vertical directions are combined to represent the ideal dog in (d).



Figure 18.5 A sampling of nine cat faces to be used in the training set for the dog versus cat recognition algorithm.

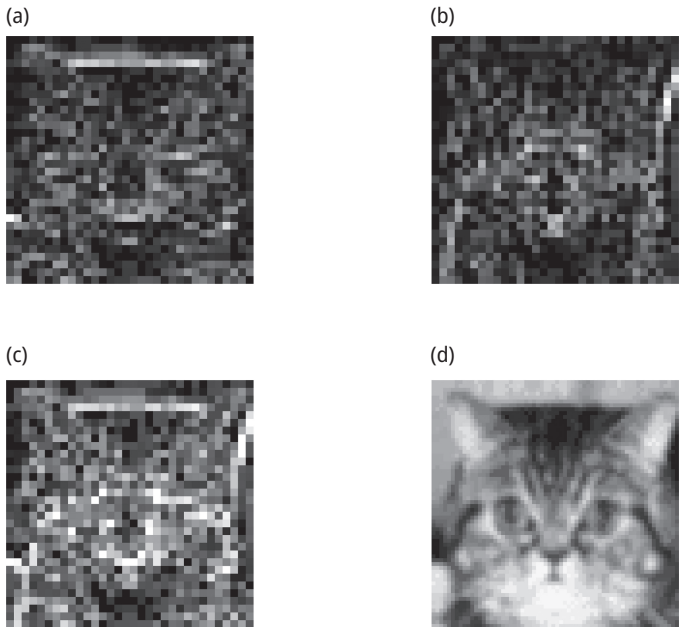


Figure 18.6 This figure represents the (a) horizontal and (b) vertical representations of the wavelet transform in the appropriately scaled image scale. The overall wavelet transformation process is represented in (c), where the edge detection in the horizontal and vertical directions are combined to represent the ideal cat in (d).

The new matrices have been appropriately scaled to the correct image brightnesses so that the images now appear sharp and clean in Fig. 18.4. Indeed, the bottom two panels of this figure show the ideal dog along with the dog represented in its wavelet basis constructed by weighting the vertical and horizontal directions.

The cat data set can similarly be analysed. Figures 18.5 and 18.6 demonstrate the wavelet decomposition of a cat. Specifically, the bottom two panels of Fig. 18.6 show the ideal cat along with the cat represented in its wavelet basis constructed by weighting the vertical and horizontal directions. Just as before, the analysis was done for the sixth cat in the data set. This choice of cat (and dog) was arbitrary, but it does illustrate the general principle quite nicely. Indeed, the bottom right figures of both Fig. 18.4 and 18.6 will form the basis of our statistical decision-making process.

18.2 The SVD and Linear Discrimination Analysis

Having decomposed the images of dogs and cats into a wavelet representation, we now turn our attention to understanding the mathematical and statistical distinction between dogs and cats. In particular, steps 2 and 3 in the image recognition algorithm of the previous section will be addressed here. In step 2, our aim will be to decompose our dog and cat

data sets via the SVD, i.e. dogs and cats will be represented by principal components or by a proper orthogonal mode decomposition. Following this step, a statistical decision based upon this representation will be made. The specific supervised learning method used here is called a *linear discrimination analysis* (LDA).

18.2.1 The SVD decomposition

The first step in setting up the statistical analysis is the SVD decomposition of the wavelet-generated data. The wavelet-transformed data is put into two data sets: **dogData_w.mat** and **catData_w.mat**, respectively.

```
dog_data = loadmat('dogData_w.mat')
dogw = dog_data['dog_wave']
cat_data = loadmat('catData_w.mat')
catw = cat_data['cat_wave']

CD = np.concatenate((dog_wave,cat_wave),axis=1)
u,s,vT = np.linalg.svd(CD-np.mean(CD),full_matrices=0)
v = vT.T
```

Recall that only a limited number of the principal components are considered in the feature detection as determined by the number of variables (or columns of **U**) that determine the feature space. Thus, upon completion of the SVD step, a limited number of columns of **U** and rows of $\mathbf{\Sigma V}^*$ are extracted. Further, the projection of each image onto this SVD feature space gives the classification space for determining dogs versus cats.

To get a better feeling of what has just happened, the SVD matrices **U**, $\mathbf{\Sigma}$, and **V** are all visualized. What is of most interest is the PCA/POD that is generated from the cat and dog data. This can be reconstructed with the following code for the jth SVD component:

```
U = np.flipud(np.reshape(u[:,j],(32,32)))
pcolor(np.rot90(U),cmap='hot')
```

Thus, the first four columns, or PCA/POD modes, are plotted by reshaping them back into 32×32 images. Figure 18.7 shows the first four POD modes associated with the dog and cat data. The strength of each of these modes in a given cat or dog can be computed by the projection onto $\mathbf{\Sigma V}^*$. In particular, the singular (diagonal) elements of $\mathbf{\Sigma}$ give the strength of the projection of each cat onto the POD modes. Figures 18.8 and 18.9 demonstrate the singular values associated with the cat and dog data along with the strength of each cat projected onto the POD modes.

Note the dominance of a single mode and the heavy-tail distribution of the remaining singular values. The singular values in Fig. 18.8 are plotted on both a normal and log scale for convenience. Figure 18.9 demonstrates the projection strength of the first 40 individual

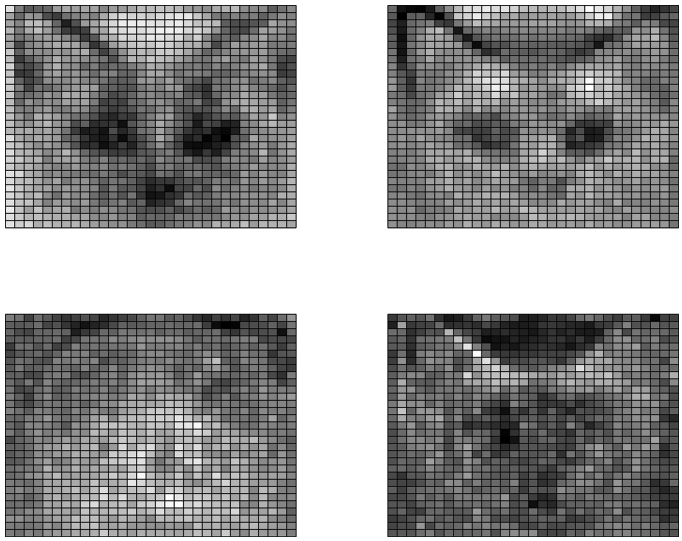


Figure 18.7 Representation of the first four principal components, or proper orthogonal modes, of the cat and dog data. Note the dominance of the triangular ears in the first two modes, suggesting a strong signature of a cat. These modes are used as the basis for classifying a cat or dog image.

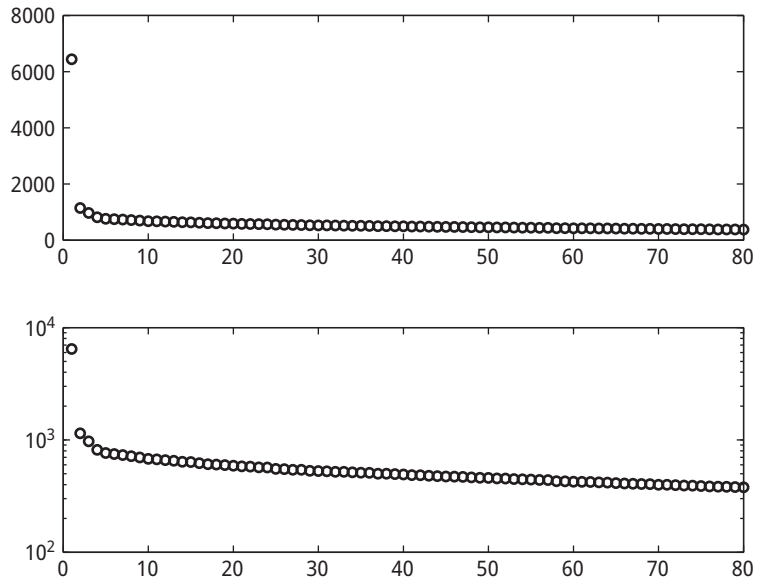


Figure 18.8 Singular values of the SVD decomposition of the dog and cat data. Note the dominance of a single singular value and the heavy-tail distribution (non-Gaussian) fall off of the singular values.

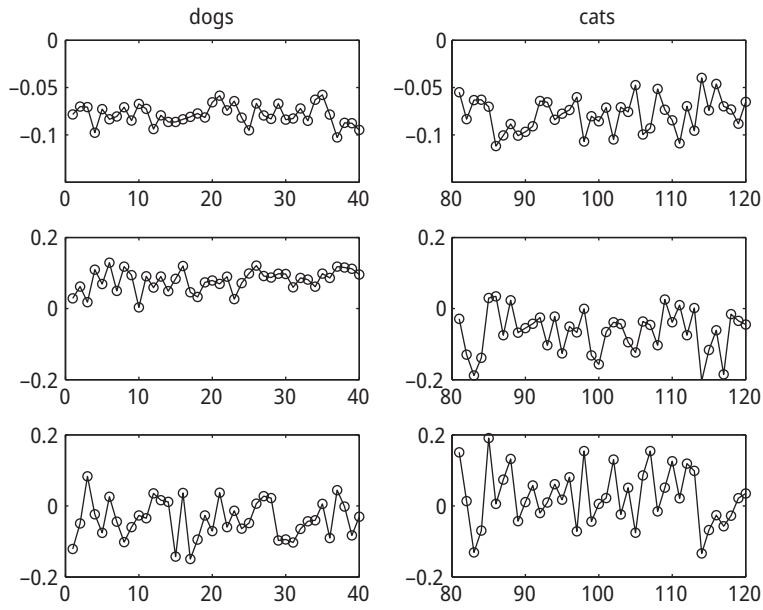


Figure 18.9 Projection of the first 40 individual cats and dogs onto the first three POD modes as described by the SVD matrix **V**. The left column is the first 40 dogs, while the right column is the first 40 cats. The three rows of figures are the projections on to the first three POD modes.

cats and dogs onto the first three POD modes. Specifically, the first three mode strengths correspond to the rows of the figures. Note the striking difference in cats and dogs as shown in the second mode. Namely, for dogs, the first and second mode are of opposite sign and "cancel," whereas for cats, the signs are the same and the features of the first and second mode add together. Thus for cats, the triangular ears are emphasized. This starts to give some indication of how the discrimination might work.

18.2.2 Linear discrimination analysis

Thus far, two decompositions have been performed: wavelets and SVD. The final piece in the training algorithm is to use statistical information about the wavelet/SVD decomposition in order to make a decision about whether the image is indeed a dog or cat. Figure 18.10 gives a cartoon of the key idea involved in the LDA. The idea of the LDA was first proposed by Fisher [144] in the context of taxonomy. In our example, two data sets are considered and projected onto new bases. In the left figure, the projection shows the data to be completely mixed, not allowing for any reasonable way to separate the data from each other. In the right figure, which is the ideal caricature for LDA, the data are well separated with the means μ_1 and μ_2 being well apart when projected onto the chosen subspace. Thus, the goal of LDA is two-fold: *find a suitable projection that maximizes the distance between the inter-class data while minimizing the intra-class data*.

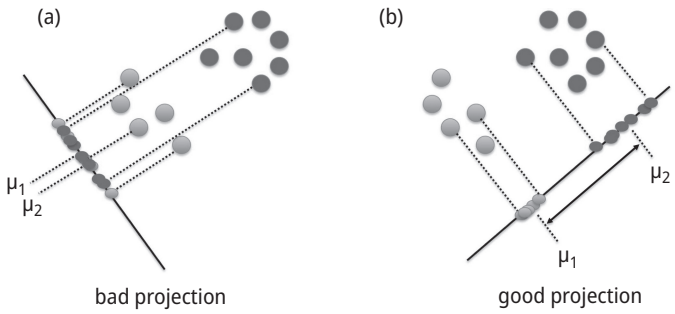


Figure 18.10 Graphical depiction of the process involved in a two-class linear discrimination analysis. Two data sets are projected onto a new basis function. In the left figure, the projection produces highly mixed data, and very little distinction or separation can be drawn between data sets. In the right figure, the projection produces the ideal, well-separated statistical distribution between the data sets. The goal is to mathematically construct the optimal projection basis, which separates the data most effectively.

For a two-class LDA, the above idea results in consideration of the following mathematical formulation. Construct a projection $\mathbf{w}$ such that

$$\mathbf{w} = \arg\max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} \tag{18.2.1}$$

where the scatter matrices for between-class $\mathbf{S}_B$ and within-class $\mathbf{S}_W$ data are given by

$$\mathbf{S}_B = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T \tag{18.2.2}$$

$$\mathbf{S}_W = \sum_{j=1}^{2} \sum_{\mathbf{x}} (\mathbf{x} - \mu_j)(\mathbf{x} - \mu_j)^T. \tag{18.2.3}$$

These quantities essentially measure the variance of the data sets as well as the variance of the difference in the means. There are a number of tutorials available online with details of the method; thus, the mathematical details will not be presented here. The criterion given by (18.2.1) is commonly known as the generalized Rayleigh quotient whose solution can be found via the generalized eigenvalue problem

$$\mathbf{S}_B \mathbf{w} = \lambda \mathbf{S}_W \mathbf{w} \tag{18.2.4}$$

where the maximum eigenvalue λ and its associated eigenvector give the quantity of interest and the projection basis. Thus, once the scatter matrices are constructed, the generalized eigenvectors can easily be constructed with Python.

Once the SVD decomposition has been performed, the LDA is applied. The following code produces the LDA projection basis as well as the decision threshold level associated with the dog and cat recognition.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

xtrain = np.concatenate((v[:60,np.array([1,3])],v[80:140,
  np.array([1,3])]))
label = np.repeat(np.array([1,-1]),60)
test = np.concatenate((v[60:80,np.array([1,3])],v[140:160,
  np.array([1,3])]))

lda = LinearDiscriminantAnalysis()
test_class = lda.fit(xtrain, label).predict(test)

truth = np.repeat(np.array([1,-1]),20)
E = 100*(1-np.sum(0.5*np.abs(test_class - truth))/40)
```

For features constructed from the first 20 SVD modes, a histogram can be constructed of the dog and cat statistics. Note that the code above only projects onto the first three components. The code not only produces a histogram of the statistical distributions associated with the dog and cat properties projected to the LDA basis, it also shows the decision threshold that is computed for the statistical decision-making problem of categorizing a given image as a dog or cat (see Fig. 18.11). Note that as is the case in most statistical processes, the threshold level shows that some cats are mistaken as dogs and vice versa. The more features that are used, the fewer "mistakes" are made in the recognition process.

Up to now, we have performed three key steps in training or organizing our dog and cat data images: we have (i) wavelet decomposed the images, (ii) considered this wavelet representation and its corresponding SVD/PCA/POD, and (iii) projected these results onto the subspace provided by the linear discrimination analysis. The three steps combined are the basis of the training set of the dog versus cat recognition problem. It remains now to

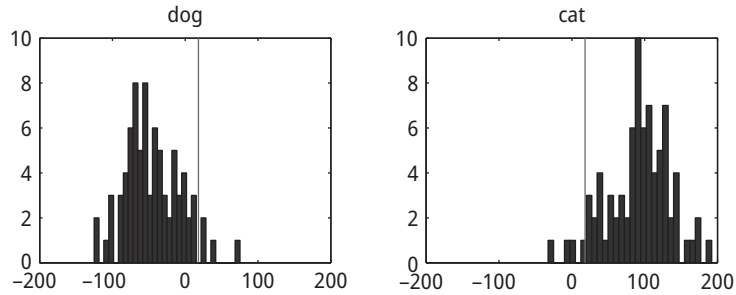


Figure 18.11 Histogram of the statistics associated with the dog and cat data once projected onto the LDA basis. The vertical line is the numerically computed decision threshold. Thus, a new image would be projected onto the LDA basis and a decision would be made concerning the image depending upon which side of the threshold line it sits.



Figure 18.12 Nine sample images of dogs and cats that will be classified by the wavelet, SVD, LDA scheme. The second dog in this set (6th picture) will be misclassified due to its ears.

test the training set (and machine learning) on a sample of dog and cat pictures outside of the training data set. The test data consists of 40 new pictures of dogs and cats (20 each) for us to test our recognition algorithm on. The test set is held out in the above code and used to predict the labels. The accuracy achieved on the test set is about 83%.

Figure 18.12 shows nine of the 40 test images of dogs and cats to be processed by our wavelet, SVD, LDA algorithm. To classify the images as dogs or cats, the new data set must now undergo the same process as the training images. Thus, the following will occur:

- Decompose the new image into the wavelet basis.
- Project the images onto the SVD/PCA/POD modes selected from the training algorithm.
- Project this two-level decomposition onto the LDA eigenvector.
- Determine the projection value relative to the LDA threshold determined in the training algorithm.
- Classify the image as dog or cat.

In addition to identifying and classifying the test data, the number of mistakes, or misidentified, images is reported and the percentage rate of success is given. In this algorithm, a nearly 83% chance of success is given using only three features (95% can be



Figure 18.13 Of the 38 images tested, two mistakes are made. The two dogs above were both misclassified as cats. Looking at the POD of the dog/cat decomposition gives a clue to why the mistake was made: they have triangular ears. Indeed, the image recognition algorithm developed here favours, in some sense, recognition of cats versus dogs simply due to the strong signature of ears that cats have. This is the primary feature picked out in the edge detection of the wavelet expansion.

achieved with 20 modes), which is quite remarkable given the low-resolution, 2D images, and the simple structure of the algorithm.

The mistaken images in the 40 test images can now be considered. In particular, by understanding why mistakes were made in identification, better algorithms can be constructed. Figure 18.13 shows the two errors that were made in the identification process. These two dogs have the quintessential features of cats: triangular ears. Recall that even in the training set, the decision threshold was set at a value for which some cats and dogs where misclassified. It is these kinds of features that can give the misleading results, suggesting that edge detection is not the only mechanism we use to identify cats versus dogs. Indeed, one can see that much more sophisticated algorithms could be used based upon the size of the animal, colour, coat pattern, walking cadence, etc. Of course, the more sophisticated the algorithm, the more computationally expensive it becomes. This is a fast, low-resolution routine that produces 95% accuracy and can be easily implemented.

18.3 Classification Trees, Support Vector Machines, and Neural Networks

To finish this chapter on supervised learning, three of the dominant paradigms of learning are considered when training labels are available: classification trees, support vector machines (SVMs), and neural networks. Before the mid-2010s, classification trees and SVMs dominated machine learning. Since then, neural networks have shown to be exceptional when provided with a large enough training data set. The underlying theory of each is presented there along with their implementation on the MNIST data set from Ch. 17. These three represent classic methods to initially try on data as they are often successful in practice.

18.3.1 Classification Trees

Leo Breiman and co-workers [85] developed the concept of the decision tree in the mid-1980s. Both implementing it in practice and establishing many of the theoretical foundations exploited today for data mining. The decision tree is a hierarchical construct that optimizes

the splitting of data in order to provide a robust classification and regression. It is the converse of a dendrogram hierarchical clustering from Ch. 17. In this case, our goal is not to move from bottom up in the clustering process, but from top down in order to create the best splits possible for classification. Having labelled data is what allows us to split the data accordingly. Classification trees gained popularity due to the fact that (i) they often produce interpretable results that can be graphically displayed, making them easy to interpret even for non-experts, (ii) they can handle numerical or categorical data equally well, (iii) they can be statistically validated so that the reliability of the model can be assessed, (iv) they perform well with large data sets at scale, and (v) the algorithms mirror human decision-making, again making them more interpretable and useful.

The long history of development for classification trees has produced a tremendous assortment of innovations, with perhaps the state-of-the-art being the XGBoost algorithm [145]. The coverage here will be limited, but we will highlight the basic architecture for data splitting and tree construction. Only the basic decision tree algorithm is considered here. It is interpretable and easy to implement. The algorithm is as follows: (i) scan through each component (feature) x_k $(k = 1, 2, \cdots, n)$ of the vector $\mathbf{x}_j$ to identify the value of x_j that gives the best labelling prediction for $\mathbf{y}_j$. (ii) Compare the prediction accuracy for each split on the feature x_j. The feature giving the best segmentation of the data is selected as the split for the tree. (iii) With the two new branches of the tree created, this process is repeated on each branch. The algorithm terminates once each individual data point is a unique cluster, known as a *leaf*, on a new branch of the tree. This is essentially the inverse of the dendrogram. Often the splits can be like coin-flips, i.e. they are very sensitive to the data in practice and different trees can be produced with small perturbations. Random forest algorithm helps stabilize this process by averaging over many perturbed decision tress [145, 146].

As with SVMs, the diversity of algorithmic innovations are vast. However, stable and easy-to-use code is available, which integrates the best algorithms available. The following code produces a classification tree for MNIST.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2)

reg_tree = DecisionTreeRegressor(random_state=42)
reg_tree.fit(X_train, y_train)

y_pred = np.round(reg_tree.predict(X_test)).astype(int)
accuracy = accuracy_score(y_test, y_pred)
plot_tree(reg_tree, feature_names=mnist.feature_names,
          class_names=mnist.target_names, filled=True)
```

This gives a tree and its predictor for new data as shown in Fig. 18.14. Moreover, the tree can actually be plotted. In the example here, only the first 300 MNIST digits are considered in order to produce a better visualization.

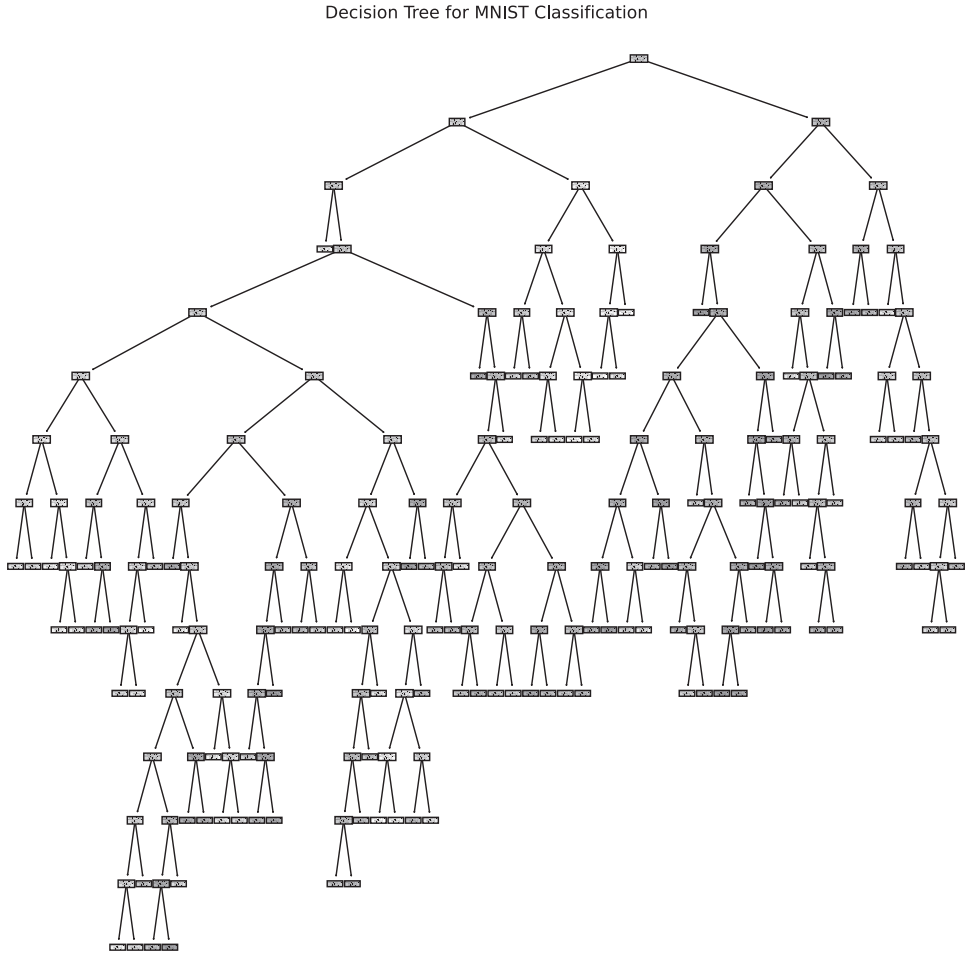


Figure 18.14 Classification tree for MNIST data for the first 300 digits. The classification tree is the opposite of a dendrogram produced in unsupervised hierarchical clustering.

18.3.2 Support Vector Machines

The origins of SVM date to 1953 from the work of Vapnik and Chervonenkis on statistical learning, where hyperplanes are optimized to split the data into distinct clusters. Nearly three decades later, Boser, Guyon, and Vapnik created nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes [147]. The current standard incarnation (soft margin) was proposed by Cortes and Vapnik in the mid-1990s [148].

The goal of the SVM method is to construct a hyperplane

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{18.3.1}$$

where the vector $\mathbf{w}$ and constant b parametrize a hyperplane that separate classes/labels in the data. The optimization of the SVM classifier attempts to make the fewest labelling errors

for the data possible while also making the largest margin between the data classes. The vectors that determine the boundaries of the margin separating the classes are termed the *support vectors*. Given the hyperplane (18.3.1), a new data point $\mathbf{x}_j$ can be classified by simply computing the sign of $(\mathbf{w} \cdot \mathbf{x}_j + b)$. Specifically, for classification labels $\mathbf{y}_j \in \{\pm 1\}$, the data to the left or right of the hyperplane is given by

$$\mathbf{y}_j(\mathbf{w} \cdot \mathbf{x}_j + b) = \operatorname{sign}(\mathbf{w} \cdot \mathbf{x}_j + b) = \begin{cases} +1 \\ -1 \end{cases} \tag{18.3.2}$$

Thus, the classifier $\mathbf{y}_j$ is explicitly dependent on the position of $\mathbf{x}_j$.

Although the linear classifier (18.3.1) is easily interpretable, linear classifiers are of limited value since they have difficulty in separating high-dimensional and complex data. Better classification curves can be constructed from the feature space of the SVM by enriching its space. SVM does this by including nonlinear features and then building hyperplanes in this new space represented as

$$\mathbf{x} \mapsto \Phi(\mathbf{x})\,. \tag{18.3.3}$$

We can call the $\Phi(\mathbf{x})$ new *observables* of the data. The SVM algorithm now learns the hyperplanes that optimally split the data into distinct clusters in a new space. Thus, one now considers the hyperplane function

$$f(\mathbf{x}) = \mathbf{w} \cdot \Phi(\mathbf{x}) + b \tag{18.3.4}$$

with corresponding labels $\mathbf{y}_j \in \{\pm 1\}$ for each point $f(\mathbf{x}_j)$. This gives a higher-dimensional space for separating the data that is directly related to Cover's theorem [149], which guarantees that the data can be separated linearly in an infinite-dimensional space.

The optimization techniques for constructing the linear SVM, nonlinear SVM, or kernel SVM are quite diverse and well developed. This is largely because the method was so successful as an algorithm for classification tasks on high-dimensional and complex data. The optimization routines are beyond the scope of this book, but we will leverage them with built-in Python packages. The following code scales the data and then applies the SVD classifier using a kernel technique.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

svm_model = SVC(kernel='rbf', random_state=42)
svm_model.fit(X_train_scaled, y_train)

y_pred = svm_model.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
```

The withheld test set is easily tested with the built-in functions in Python. A good way to think about SVM: it is the natural and nonlinear generalization of linear discriminant analysis of Sec. 18.2. In fact, many of the figures of the last section concerning LDA apply here except that the space on which they act is much higher-dimensional.

18.3.3 Neural Networks

Finally, we consider neural networks. Specifically, we consider the general mapping as given in Ch. 6. The mapping is then a fit from inputs to targets, which are the labels

$$\mathbf{Y} = f_{\boldsymbol{\theta}}(\mathbf{X}) \tag{18.3.5}$$

where the parameters $\boldsymbol{\theta}$ are found by optimization to a *goodness-of-fit* of this function to data. Specifically, the goodness-of-fit is the number of labels correctly captured in the model upon training and cross-validation. For the case of a MNIST, the input $\mathbf{X}$ is an image of digit, and the output is its label $\mathbf{Y}$, which goes from 0 to 9. With enough training data, the hope is to optimize the parameters $\boldsymbol{\theta}$ so that a minimal number of mistakes are made in the model on a withheld test set.

We construct a simple feed-forward neural network that has three layers. The input to the network is a flattened version of the 28 by 28 image. Thus, the input vectors $\mathbf{x}_k$ comprising the input data $\mathbf{X}$ are of size $28 \times 28 = 784$. The output is an *indicator* vector $\mathbf{y}$ whose first component is nonzero for digit 0, its second component is nonzero for a digit 1, and so forth. The neural network is specified in PyTorch as follows:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784) # flatten the input image
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Note that the activation layers are ReLU in the first and second layers. A diversity of architectures and activation functions can be used to potentially improve performance. This is a very simple example to illustrate how one might construct a feed-forward network mapping an input to a classification label.

The data can be split into a training and test set for ingestion into PyTorch. The following code trains the feed-forward neural network using stochastic gradient descent using a cross entropy for loss. Both optimizers and loss functions can be traded out easily in the PyTorch environment.

```
train_loader = torch.utils.data.DataLoader
   (dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader
   (dataset=test_dataset, batch_size=64, shuffle=False)

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

num_epochs = 10
     for epoch in range(num_epochs):
         for i, (images, labels) in enumerate(train_loader):
             optimizer.zero_grad()
             outputs = net(images)
             loss = criterion(outputs, labels)
             loss.backward()
             optimizer.step()

with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

The code shows a number of hyper-parameters, which can be tuned for potentially better behaviour. Specifically, the learning rate, the batch size, the optimizer, the loss function, the neural network architecture (layers and sizes of layers) can all be modified to enhance performance. Hyper-parameter tuning is generally an expensive and time-intensive exercise.

19 Reinforcement Learning

19.1 Mathematical Architecture of Reinforcement Learning 514
19.1.1 Agent State Dynamics 517
19.2 Markov Decision Process for Reinforcement Learning 518
19.3 Policy Optimization 520
19.3.1 The snake–apple game 521

Machine learning provides a set of mathematical optimization tools for extracting and leveraging features from data. In unsupervised and supervised learning, many of these efforts have centred around providing classification and clustering of data. The use of such methods are diverse and extensive, spanning application areas such as signal processing and computer vision. Reinforcement learning is altogether different and centres much more strongly on what might be considered machine intelligence and AI. Reinforcement learning (RL) is a machine learning architecture that interacts and learns from the environment in which it is embedded, making it a powerful paradigm for technological integration. In this chapter, a basic summary of RL is given. For a more detailed and compressive treatment, we refer the reader to the classic text [150].

19.1 Mathematical Architecture of Reinforcement Learning

Reinforcement learning is a mathematical architecture allowing for machine learning to interact with its environment, whether virtual or real. The greatest successes of RL to date have come from virtual environments where it has been shown to have unmatched capabilities. This includes AlphaGo and AlphaGo Zero from DeepMind, which are algorithms that have mastered the game of Go beyond any human players. Although successful in virtual environments, RL has largely failed to transition into practical engineering problems. This is in large part due to how it is trained. Specifically, in real world environments, RL is impractical to implement, as it relies on exploration of all possibilities for achieving its goal.

To make the RL architecture more concrete, we begin with basic concepts for framing the mathematical structure

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0020

- **Agent:** The agent is the embodiment of the entity that will be trained to interact with the environment. In a physical world, the agent is a robot, for instance. In a game like Go or chess, the agent is the player. The agent is the entity, which is trained to embody the machine intelligence or AI.
- **Environment:** The agent is embedded in an environment. The environment can be virtual, as in gameplay, or real, as when robots interact within a physical setting. The environment itself can be changing with time and location. Ultimately, the agent learns to navigate its environment in order to best optimize or achieve a prescribed goal.
- **Observations:** In order for the agent to navigate the environment, it must be capable of taking measurements of the environment. In the physical world, observations would come from a diversity of sensors. In a virtual world, aspects of the environment would be given to the agent over time.
- **Actions:** The agent is endowed with the capability of taking actions in the environment given a set of observations. The action space is typically limited by physical constraints. In gameplay, actions are limited by the rules of the game in which the agent is participating in. The goal of RL is to train the actions of the agent in order to best achieve an objective.

The concepts frame the mathematical architecture of RL. Moreover, they endow the agent with human-like quality of learning, whereby the agent interacts with the environment in order to effectively learn strategies for accomplishing a goal.

The name reinforcement learning is derived from the field of psychology. Specifically, learning is achieved by reinforcing a concept through reward. As such, there is only one last thing missing: a reward function. Fundamental to RL is the reward hypothesis, which is stated as follows:

> **Reward Hypothesis:** Any goal can be formalized as the outcome of maximizing a cumulative reward.

It is upon the hypothesis that the RL agent acquires machine intelligence.

With the definitions above, a mathematical representation of RL can be posited. Specifically, the following key variables are defined:

$$\mathcal{S}_t - \text{State of the environment at time } t \tag{19.1.1a}$$

$$\mathcal{O}_t - \text{Observations of the environment at time } t \tag{19.1.1b}$$

$$\mathcal{A}_t - \text{Actions at time } t \tag{19.1.1c}$$

$$\mathcal{R}_t - \text{Reward (scalar) received at time } t. \tag{19.1.1d}$$

Within this mathematical framing, one can also assess whether a system if fully observable, i.e. $\mathcal{O}_t = \mathcal{S}_t$, or only partially observable, whereby $\mathcal{O}_t \neq \mathcal{S}_t$. Games such as chess and Go are fully observable systems. Physical systems, however, are largely only partially observable. Figure 19.1 shows the basic structure of the RL strategy.

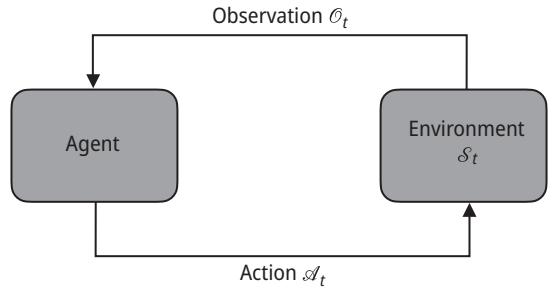


Figure 19.1 General structure of the reinforcement learning architecture. The agent samples from the environment $\mathcal{S}_t$ via observations $\mathcal{O}_t$. The agent then produces actions $\mathcal{A}_t$, which produce new observations through interactions with the environment.

The basic idea of RL as an algorithm is as follows:

Step 1: Receive observations $\mathcal{O}_t$ and reward $\mathcal{R}_t$ at time t
Step 2: Execute an action $\mathcal{A}_t$ to maximize the reward
Repeat: Given the action, now receive $\mathcal{O}_{t+1}$ and $\mathcal{R}_{t+1}$ and repeat

This shows the algorithm to be iterative in nature. Thus, a sequence of actions $\mathcal{A}_t$ are to be taken to maximize the cumulative reward, not just the immediate reward $\mathcal{R}_t$ or $\mathcal{R}_{t+1}$.

Given the ability to compute an instantaneous scalar reward $\mathcal{R}_t$, RL seeks to maximize the cumulative reward for the future. This assumes causality, which allows the possible actions to only influence the future state of the system. The cumulative future reward is then given by

$$\mathcal{G}_t = \mathcal{R}_{t+1} + \mathcal{R}_{t+2} + \mathcal{R}_{t+3} + \cdots . \tag{19.1.2}$$

This gives a framework allowing for the optimization of actions now and in the future in order to generate the greatest rewards. In fact, within this framework, we can optimize the expected return, or value, given by

$$v(\mathcal{S}) = E\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}\right] = E\left[\mathcal{R}_{t+1} + \mathcal{R}_{t+2} + \cdots | \mathcal{S}_t = \mathcal{S}\right]. \tag{19.1.3}$$

This formulates the value function as a probabilistic function. Thus, the value function is evaluated given the condition that the state of the system at time t is $\mathcal{S}_t = \mathcal{S}$. The causality of the value function thus estimates the future given that the system is in a prescribed state. Importantly, there is a recursive property to the cumulative reward and value function

$$\mathcal{G}_t = \mathcal{R}_{t+1} + \mathcal{G}_{t+1} \tag{19.1.4a}$$

$$v(\mathcal{S}) = E\left[\mathcal{R}_{t+1} + v(\mathcal{S}_{t+1}) | \mathcal{S}_t = \mathcal{S}\right]. \tag{19.1.4b}$$

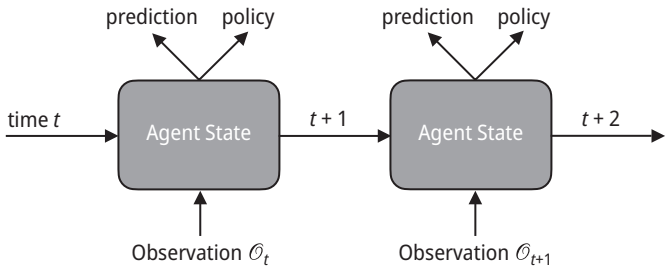


Figure 19.2 The state space model for the agent. The agent is given a policy for predictions of the dynamic model given a set of observations $\mathcal{O}_t$.

The value of an action is given by the following:

$$q(S, a) = E\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}, \mathcal{A}_t = a\right] \tag{19.1.5}$$

where a is a specific action that is considered for $\mathcal{A}_t$.

19.1.1 Agent State Dynamics

The agent is assigned to make decisions within the mathematical architecture described above. Specifically, the agent interacts with an environment $\mathcal{S}_t$, which it observes through $\mathcal{O}_t$. The range of actions are given by a, with an action $\mathcal{A}_t$ taken at a given time t. The reward $\mathcal{R}_t$ for the agent is prescribed by an end user. It then attempts to take a sequence of actions $\mathcal{A}_t$ in time which can increase its overall cumulative reward. Individual potential actions a can be valued through $q(\mathcal{S}, a)$. Figure 19.2 shows the basic structure of the state-space models.

Within this construct, the agent can be characterized by a state space. The state-space dynamics from its initial starting time to the present is given by

$$H_t = \mathcal{O}_0\mathcal{A}_0\mathcal{R}_1, \mathcal{O}_1\mathcal{A}_1\mathcal{R}_2, \cdots, \mathcal{O}_{t-1}\mathcal{A}_{t-1}\mathcal{R}_t, \mathcal{O}_t \tag{19.1.6}$$

with the agent state being prescribed by the update function of its history

$$\mathcal{S}_{t+1} = u(\mathcal{S}_t\mathcal{A}_t, \mathcal{R}_{t+1}, \mathcal{O}_{t+1}). \tag{19.1.7}$$

Given its state space and supporting variables, the agent then has a policy for defining its behaviour within the environment. The policy relates the state of the system $\mathcal{S}$ to the action

$$\mathcal{A} = \pi(\mathcal{S}) \tag{19.1.8}$$

or alternatively $\pi(\mathcal{A}|\mathcal{S}) = p(\mathcal{A}|\mathcal{S})$. Again, this is defined in conditional probability terms where one determines the probability of an action $\mathcal{A}$ given the state of the system $\mathcal{S}$. The agent value can be then finally computed as

$$v_\pi(\mathcal{S}) = E\left[\mathcal{G}_t | \mathcal{S}_t = S, \pi\right] \tag{19.1.9}$$

where the value function is explicitly conditioned on the policy of the agent as well as the state of the system. More broadly, this can be replaced with

$$v_\pi(\mathcal{S}) = E\left[\mathcal{R}_{t+1} + \gamma\mathcal{R}_{t+2} + \gamma^2\mathcal{R}_{t+3}\cdots|\mathcal{S}_t = S, \pi\right] \tag{19.1.10}$$

where $\gamma \in [0, 1]$ is a discount factor. The discount factor devalues very long time rewards for more immediate rewards. Specifically if $\gamma = 0$, the agent acts only to maximize immediate rewards without concern for long-term cumulative gains. As $\gamma \to 1$, the agent values the long-term cumulative rewards so that the immediate action may not optimize the value function. Typically, a balance between short-term and long-term rewards is achieved by the parameter γ.

19.2 Markov Decision Process for Reinforcement Learning

One of the simplest RL models is structured mathematically around a Markov model. A Markov decision process model is based on fairly simple linear algebra techniques. To construct the model, we consider first the Markov property, which states that future only depends upon the current time. Mathematically, we can consider this in conditional probability terms

$$P_{\mathcal{S}\mathcal{S}'} = P\left[\mathcal{S}_{t+1} = \mathcal{S}'|\mathcal{S}_t = \mathcal{S}\right] = P\left[\mathcal{S}_{t+1} = \mathcal{S}'|\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \cdots, \mathcal{S}_t\right] \tag{19.2.1}$$

where in the last expression, the conditional dependence is shown for the entire state time history. But since this is equal to the conditional when only $\mathcal{S}_t$ is included, it shows that the previous times do not contribute. This Markov property is used in what follows to generate a simplified model for RL.

To proceed, we consider a system that has n states. The probability of transitioning between states is given by a Markov transition matrix

$$\mathbf{P} = \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1n} \\ P_{21} & P_{22} & \cdots & P_{2n} \\ \vdots & & & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{nn} \end{bmatrix} \tag{19.2.2}$$

where P_{jk} determines the probability of moving from the jth state to the kth state. To specify the state of the system and the actions to be taken, we parametrize the model with the following:

$$(\mathcal{S}, \mathbf{P}, \mathcal{R}, \gamma) \to (\text{state, transitions, reward, discount}). \tag{19.2.3}$$

These variables must then be framed in terms of the reward function

$$\mathcal{R}_\mathcal{S} = E\left[\mathcal{R}_{t+1}|\mathcal{S}_t = \mathcal{S}\right] \tag{19.2.4}$$

with the total reward being given by

$$\mathcal{G}_t = \mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} \tag{19.2.5}$$

where the discount $\gamma \in [0, 1]$. The discount parameter is quite important in practice. Short-term rewards are often preferred. In practice, it helps with placing so much reward into an uncertain future. Thus, uncertainty in the model does not create such poor predictions for a long-term future. In practice, it also keeps the reward structure bounded.

We can now consider the value function and how to optimize it. Recall that we aim to maximize the value function

$$v(\mathcal{S}) = E\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}\right]. \tag{19.2.6}$$

The value function is evaluated over different trajectories of the future in order to assess the most valuable trajectory. The value function is averaged in order to compute an expected return.

Using recursive relationships, we can derive a Bellman equation. Specifically, consider

$$\begin{aligned} v(\mathcal{S}) =& E\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}\right] \\ =& E\left[\mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \cdots | \mathcal{S}_t = \mathcal{S}\right] \\ =& E\left[\mathcal{R}_{t+1} + \gamma \left(\mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \cdots\right) | \mathcal{S}_t = \mathcal{S}\right] \\ =& E\left[\mathcal{R}_{t+1} + \gamma \mathcal{G}_{t+1} | \mathcal{S}_t = \mathcal{S}\right] \\ =& E\left[\mathcal{R}_{t+1} + \gamma v(\mathcal{S}_{t+1}) | \mathcal{S}_t = \mathcal{S}\right] \end{aligned}$$

which gives the immediate reward along the value of a new state. The last expression is a Bellman equation for recursively estimating the value function. More succinctly, this can be expressed as

$$v(\mathcal{S}) = \mathcal{R}_{\mathcal{S}} + \gamma \sum_{\mathcal{S}' \in \mathcal{S}} P_{\mathcal{S}\mathcal{S}'} v(\mathcal{S}') \tag{19.2.7}$$

or in matrix form

$$\mathbf{v} = \mathbf{R} + \gamma \mathbf{P} \mathbf{v} \tag{19.2.8}$$

where the vectors $\mathbf{v}$ and $\mathbf{R}$ are n-dimensional, while $\mathbf{P}$ is an $n \times n$ matrix. The solution for this linear system can be found in theory from simple matrix inversion

$$\mathbf{v} = \left(\mathbf{I} - \gamma \mathbf{P}\right)^{-1} \mathbf{R} \tag{19.2.9}$$

which appears simple enough to solve. However, in practice, the n-dimensional state space of possible trajectories makes the solution computationally intractable.

For RL, the only thing missing in the Markov decision process is the effect of an action. Thus, the representation of the system is modified to include actions

$$(\mathcal{S}, \mathcal{A}, \mathbf{P}, \mathcal{R}, \gamma) \rightarrow (\text{state, action, transitions, reward, discount}) \tag{19.2.10}$$

where $\mathcal{A}$ is now a finite set of actions. Transition probabilities now include the action

$$P^{a}_{\mathcal{S}\mathcal{S}'} = P\left[\mathcal{S}_{t+1} = \mathcal{S}' | \mathcal{S}_t = \mathcal{S}, \mathcal{A}_t = a\right] \tag{19.2.11}$$

and the reward function now explicitly includes the actions

$$\mathcal{R}^{a}_{\mathcal{S}} = E\left[\mathcal{R}_{t+1} | \mathcal{S}_t = \mathcal{S}, \mathcal{A}_t = a\right]. \tag{19.2.12}$$

Despite the inclusion of the action, everything remains the same in the Bellman equation.

However, we now define the policy of the system

$$\pi(a|\mathcal{S}) = P\left[\mathcal{A}_t = a | \mathcal{S}_t = \mathcal{S}\right] \tag{19.2.13}$$

which ultimately defines the behaviour of the agent. The policy is a distribution over actions given the state of the system. Thus, the policy is conditioned on the current state.

For simplicity in this analysis, we consider stationary statistics, whereby the distributions remain constant over time. This allows for the construction of the probability transition matrix, which includes the policy

$$P^{\pi}_{\mathcal{S}\mathcal{S}'} = \sum \pi(a|\mathcal{S}) P^{a}_{\mathcal{S}\mathcal{S}'}. \tag{19.2.14}$$

The coefficient $\pi(a|\mathcal{S})$ gives the average of the transition given the policy, which is critically important for computing the overall value function.

The value of the policy can also be evaluated using

$$v_{\pi}(\mathcal{S}) = E_{\pi}\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}\right]. \tag{19.2.15}$$

The value of a specific action can be further computed from $q_{\pi}(\mathcal{S}, a)$ which starts from state $\mathcal{S}$ and takes action a. This is given by

$$q_{\pi}(\mathcal{S}, a) = E_{\pi}\left[\mathcal{G}_t | \mathcal{S}_t = \mathcal{S}, \mathcal{A}_t = a\right] \tag{19.2.16}$$

which gives the expected total reward. Computing these quantities can be exceptionally difficult in practice. Thus, the simplifications to the Bellman equation is important, as it gives a viable path for evaluation. Alternatively, RL is trained at scale by large tech companies who can afford to compute or approximate many of the quantities of interest in RL applications, both virtual such as chess or Go, or in the physical world like robotics. Modern RL methods seek to impose constraints, such as physics-informed, in order to make RL training more tractable.

19.3 **Policy Optimization**

A central goal in RL is to optimize the policy of the agent in order to generate the largest cumulative reward. This then comes down to the consideration of

$$\pi(a|\mathcal{S}) = P\left[\mathcal{A}_t = a|\mathcal{S}_t = \mathcal{S}\right] \tag{19.3.1}$$

where

$$a \in \mathcal{A} \tag{19.3.2}$$

and a is an action and $\mathcal{A}$ is all possible actions. For the Bellman equation of the previous section, the optimization lead to a value solution $\mathbf{v} = (\mathbf{I} - \gamma\mathbf{P})^{-1}\mathbf{R}$. This was intractable computationally for very large action and state spaces. For instance, chess or Go have exceptionally high-dimensional state spaces, which would be computationally intractable to solve the Bellman equation for.

An alternative for solving the Bellman equation is by leveraging dynamic programming. This can be constructed by formulating an iteration

$$v_{k+1} = E\left[\mathcal{R}_{t+1} + \gamma v_k(\mathcal{S}_{t+1})|\mathcal{S}_t = \mathcal{S}\right]. \tag{19.3.3}$$

The dynamic program then iterates to a fixed-point solution. Alternatively, one can produce a policy improvement through iteration, whereby

$$v_{\pi'}(\mathcal{S}) \geq v_\pi(\mathcal{S}). \tag{19.3.4}$$

The policy improvement theorem looks to update the policy via optimization so that

$$\pi'(\mathcal{S}) = \max_a q_\pi(\mathcal{S}, a). \tag{19.3.5}$$

The optimization thus looks for the best improvement of the policy from $\mathcal{S}$ to $\mathcal{S}'$. Dynamic programming is a well-established optimization method to perform such calculations, and they are used in practice to update the policy. In general, one learns quickly when doing any form of machine learning that optimization at scale is critical for enabling unsupervised and supervised machine learning, deep learning, and reinforcement learning. The variety of optimization methods are not considered here as we have only highlighted the very basic features of RL, which is significantly advanced beyond this in terms of sophistication, implementation, and architectures.

19.3.1 The snake–apple game

RL is often trained in what are called *gym* environments. Training gyms are simulation engines, which emulate either a true physical system environment, or the gym is a game (e.g. chess or Go). The gym allows an agent to interact with the virtual environment through simulation. Ultimately, this allows the agent to learn an optimal policy $\pi(\mathcal{S})$.

As a specific example for the use of RL, we develop a snake–apple game. The game is played on a specified grid. The game can be cloned from the Python repository from Vincent van Wynendaele:

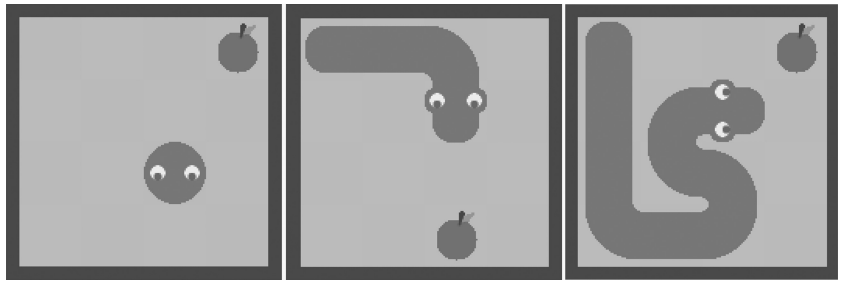


Figure 19.3 Gameplay for the snake–apple game. The snake is constrained to a grid. Touching the walls or itself results in death. There is a reward for eating the apple, which grows the worm by one unit. Winning the game means growing to full length.

```
https://github.com/Vinwcent/SnakeReinf
```

The snake starts with unit length (a worm which is the length of a single grid cell). The goal is for the worm to learn to move towards the apple and eat it. The worm is rewarded and grows by one unit length. The long-term reward is for winning the game, which means that the worm has become full length and occupies all cells. To structure the gameplay, the RL agent is allowed to take actions: moving right, moving left, moving up, and moving down. Figure 19.3 shows a series of snapshots as the game progresses.

Parameters of the gameplay are established before training, including the grid size and the reward structure. The file **launch.py** contains the key element settings for the game. The following code blocks specify how the game is played out:

```
from game.Snake import Snake
from reinf.SnakeEnv import SnakeEnv
from reinf.utils import perform_mc, show_games

# Winning everytime hyperparameters
grid_length = 4  # box size
n_episodes = 10000  # number of trials run
epsilon = 0.1 #  exploratory behavior (=0 no randomness, =1
    totally random)
gamma = 0.98 #  discount for future rewards
rewards = [-2, -1, 5, 1000] # see below
# [Losing move, inefficient move, efficient move, winning
   move]
```

The key settings are the parameter epsilon $\in [0, 1]$, which determines the degree of exploration, the parameter gamma $\in [0, 1]$, which determines the discounted future reward, and

the reward for a losing move, inefficient move, efficient move, and winning move. The winning move is when the snake wins the game and becomes full length.

The game can be played interactively.

```
game = Snake((800, 800), grid_length)
game.start_interactive_game()
```

Alternatively, one can run thousands, or hundreds of thousands of games in the gym environment for training. Specifically, the games generate a policy optimization during gameplay.

```
# Training part
env = SnakeEnv(grid_length=grid_length, with_rendering=False)
q_table = perform_mc(env, n_episodes, epsilon, gamma, rewards)
```

Once trained, the game can be played with the learned policy and visualized.

```
# Viz part (new games)
env = SnakeEnv(grid_length=grid_length, with_rendering=True)
show_games(env, 20, q_table)
```

In the above, 20 games are played using the trained q-table from the RL training module. By tuning the parameters and playing enough games, one can teach the RL to win the game with high probability. This is left as an exercise for this chapter.

20 Spatio-Temporal Data and Dynamics

20.1 Modal Expansion Techniques for PDEs 524
20.2 PDE Dynamics in the Right (Best) Basis 528
20.2.1 $N = 1$ soliton reduction 529
20.2.2 $N = 2$ soliton reduction 531
20.3 The POD Method and Symmetries/Invariances 533
20.3.1 Advection–Diffusion equations 536
20.4 POD Using Robust PCA 540
20.4.1 POD mode selection via robust PCA 541
20.5 Shallow Recurrent Decoders (SHRED) for Sensing and PDEs 544
20.6 Sparse Identification of Nonlinear Dynamics (SINDy) 549
20.7 Deep Learning Paradigms for Time–Space Stepping 554

One of the most important uses of computational methods and techniques is in its applications to modelling of physical, engineering, or biological systems that demonstrate spatio-temporal dynamics and patterns. For instance, the field of fluid dynamics fundamentally revolves around being able to predict the time and space dynamics of a fluid through some potentially complicated geometry. Understanding the interplay of spatial patterns in time is indeed often the central focus of the field of partial differential equations. In systems where the underlying spatial patterns exhibit low-dimensional (pattern forming) dynamics, the application of the proper orthogonal decomposition (POD) can play a critical role in predicting the resulting low-dimensional dynamics.

20.1 Modal Expansion Techniques for PDEs

Partial differential equations (PDEs) are fundamental to the understanding of the underlying physics, biophysics, etc. of complex dynamical systems. In particular, a PDE gives the governing evolution of a system in both time and space. Abstractly, this can be represented by the following PDE system:

$$\mathbf{U}_t = N(\mathbf{U}, \mathbf{U}_x, \mathbf{U}_{xx}, \cdots, x, t) \tag{20.1.1}$$

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0021

where $\mathbf{U}$ is a vector of physically relevant quantities, and the subscripts t and x denote partial differentiation. The function $N(\cdot)$ captures the space–time dynamics that is specific to the system being considered. Along with this PDE are some prescribed boundary conditions and initial conditions. In general, this can be a complicated, nonlinear function of the quantity $\mathbf{U}$, its derivatives, and both space and time.

In general, there are no techniques for building analytic solutions for (20.1.1). However for specific forms of $N(\cdot)$, such as the case where the function is linear in $\mathbf{U}$, time-independent, and constant coefficient, the standard PDE solution techniques may be used to characterize the dynamics analytically. Such solution methods are the core of typical PDE courses at both the graduate and undergraduate level. The specific process for solving PDE systems involves reduction from a PDE to ODE, and then a reduction from an ODE to algebra is enacted. Provided such reductions can be achieved and undone, then an analytic solution is produced.

A number of standard solution techniques can be discussed. The first to be highlighted is the technique of self-similarity reduction. This method simply attempts to extract a fundamental relationship between time and space by making a change of variable to, for instance, a new variable such as

$$\xi = x^{\beta} t^{\alpha} \tag{20.1.2}$$

where β and α are determined by making the change of variable in (20.1.1). This effectively performs the first step in the reduction process by reducing the PDE to an ODE in the variable ξ alone. A second, and perhaps the most common introductory technique, is the method of separation of variables where the assumption

$$\mathbf{U}(x,t) = \mathbf{F}(x)\mathbf{G}(t) \tag{20.1.3}$$

is made, thus effectively separating time and space in the solution. When applied to (20.1.1) with $N(\cdot)$ being linear in $\mathbf{U}$, two ODEs result: one for the time dynamics $\mathbf{G}(t)$ and one for the spatial dynamics $\mathbf{F}(x)$. If the function $N(\cdot)$ is nonlinear in $\mathbf{U}$, then separation cannot be achieved.

As a specific example of the application of the method of separation, we consider here the eigenfunction expansion technique. Strictly speaking, to make analytic progress with the method of separation of variables, one must require that separation occurs. However, no such requirement needs to be made if we are simply using the method as the basis of a computational technique. In particular, the eigenfunction expansion technique assumes a solution of the form

$$u(x,t) = \sum_{n=1}^{\infty} a_n(t)\phi_n(x) \tag{20.1.4}$$

where $\phi_n(x)$ is an orthogonal set of eigenfunctions, and we have assumed that the PDE is now described by a scalar quantity $u(x,t)$. The $\phi_n(x)$ can be any orthogonal set of functions such that

$$(\phi_j(x), \phi_k(x)) = \delta_{jk} = \begin{cases} 1 & j = k \\ 0 & j \neq q \end{cases} \tag{20.1.5}$$

where δ_{jk} is the Dirac function and the notation $(\phi_j, \phi_k) = \int \phi_j \phi_k^* dx$ gives the inner product. For a given physical problem, one may be motivated to use a specified set of eigenfunctions such as those special functions that arise for specific geometries or symmetries. More generally, for computational purposes, it is desired to use a set of eigenfunctions that produce accurate and rapid evaluation of the solutions of (20.1.1). Two eigenfunctions immediately come to mind: the Fourier modes and Chebyshev polynomials. This is largely in part due to their spectral accuracy properties and tremendous speed advantages such as $O(N \log N)$ speed in projection to the spectral basis.

To give a specific demonstration of this technique, consider the nonlinear Schrödinger (NLS) equation

$$iu_t + \frac{1}{2}u_{xx} + |u|^2 u = 0 \tag{20.1.6}$$

with the boundary conditions $u \to 0$ as $x \to \pm\infty$. If not for the nonlinear term, this equation could be solved easily in closed form. However, the nonlinearity mixes the eigenfunction components in the expansion (20.1.4) making a simple analytic solution not possible.

To solve the NLS computationally, a Fourier mode expansion is used. Thus, use can be made of the standard fast Fourier transform. The following code formulates the PDE solution as an eigenfunction expansion technique (20.1.4) of the NLS (20.2.2). The first step in the process is to define an appropriate spatial and time domain for the solution along with the Fourier frequencies present in the system. The following code produces both the time and space domain of interest:

```
L = 40; n = 512
x2 = np.linspace(-L/2, L/2, n+1); x = x2[:n]
k = (2*np.pi/L) * np.concatenate((np.arange(0, n//2),
np.arange(-n//2, 0)))
t = np.arange(0, 10.1, 0.1)
```

It now remains to consider a specific spatial configuration for the initial condition. For the NLS, there are a set of special initial conditions called solitons, where the initial conditions are given by

$$u(x, 0) = N\text{sech}(x) \tag{20.1.7}$$

where N is an integer. We will consider the soliton dynamics with $N = 1$ and $N = 2$, respectively. In order to do so, the initial condition is projected onto the Fourier modes with the fast Fourier transform. Rewriting (20.2.2) in the Fourier domain, i.e. Fourier transforming, gives the set of differential equations

$$\hat{u}_t = -\frac{i}{2}k^2\hat{u} + i\widehat{|u|^2 u} \tag{20.1.8}$$

where the Fourier mode mixing occurs due to the nonlinear mixing in the cubic term. This gives the system of differential equations to be solved for in order to evaluate the NLS behaviour. The following code solves the set of differential equations in the Fourier domain.

```
def nls_rhs(ut0, t, k, n):
    ut= ut0[0:n] + 1j*ut0[n:]
    u = np.fft.ifft(ut)
    rhs = -(1j/2) * (k**2) * ut + 1j * np.fft.fft((np.abs(u)**2) * u)
    return np.hstack([np.real(rhs),np.imag(rhs)])

N = 1; u = N * np.cosh(x)**(-1); ut = np.fft.fft(u)
ut0= np.hstack([np.real(ut),np.imag(ut)])

utsol = odeint(nls_rhs, ut0, t, args=(k,n,))
usol = np.zeros((len(t), n), dtype=np.complex128)
usol2 = utsol[:,0:n] + 1j*utsol[:,n:]

for j in range(len(t)):
    usol[j, :] = np.fft.ifft(usol2[j, :])
```

This gives a complete code that produces both the time–space evolution and the time–frequency evolution.

The dynamics of the $N = 1$ and $N = 2$ solitons are demonstrated in Figs. 20.1 and 20.2, respectively. During evolution, the $N = 1$ soliton only undergoes phase changes, while its amplitude remains stationary. In contrast, the $N = 2$ soliton undergoes periodic oscillations. In both cases, a large number of Fourier modes, about 50 and 200, respectively, are required to model the simple behaviours illustrated.

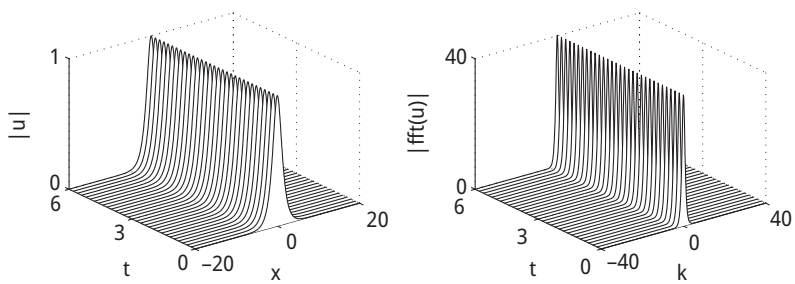


Figure 20.1 Evolution of the $N = 1$ soliton. Although it appears to be a very simple evolution, the phase is evolving in a nonlinear fashion, and approximately 50 Fourier modes are required to model this behaviour accurately.

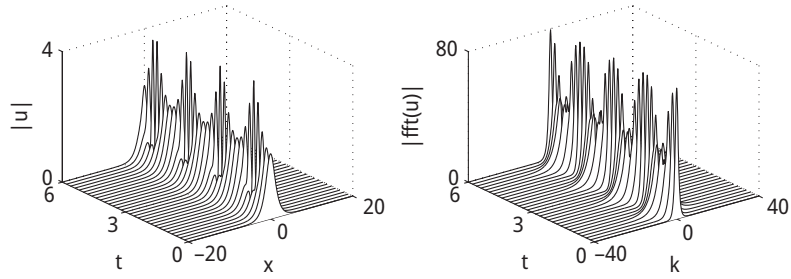


Figure 20.2 Evolution of the $N = 2$ soliton. Here a periodic dynamics is observed, and approximately 200 Fourier modes are required to model the behaviour accurately.

The potentially obvious question to ask in light of our dimensionality reduction thinking is this: is the soliton dynamics really a 50 or 200 degree-of-freedom system as implied by the Fourier mode solution technique? The answer is no. Indeed, with the appropriate basis, i.e. the POD modes generated from the singular value decomposition (SVD), it can be shown that the dynamics is a simple reduction to one or two modes, respectively. Indeed, it can easily be shown that the $N = 1$ and $N = 2$ are truly low-dimensional by computing the SVD of the evolutions shown in Figs. 20.1 and 20.2, respectively. Indeed, just as in the example of Sec. 15.5, the simulations can be aligned in a matrix $\mathbf{A}$, where the columns are different segments of time and the rows are the spatial locations.

Figures 20.3 and 20.4 demonstrate the low-dimensional nature explicitly by computing the singular values of the numerical simulations along with the modes to be used in our new eigenfunction expansion. What is clear is that for both of these cases, the dynamics is truly low-dimensional with the $N = 1$ soliton being modelled exceptionally well by a single POD mode, while the $N = 2$ dynamics is modelled quite well with two POD modes. Thus, in performing the expansion (20.1.4), the modes chosen should be the POD modes generated from the simulations themselves. Quite complicated models can be considered that produce low-dimensional behaviour [151]. In the next section, we will derive the dynamics of the modal interaction for these two cases and show that quite a bit of analytic progress can then be made within the POD framework.

20.2 PDE Dynamics in the Right (Best) Basis

The last section demonstrated the $N = 1$ and $N = 2$ soliton dynamics of the NLS equation (20.2.2). In particular, Figs. 20.1 and 20.2 showed the evolution dynamics in the Fourier mode basis that was used in the computation of the evolution. The primary motivation in performing a POD reduction of the dynamics through (20.1.4) is the observation in Figs. 20.3 and 20.4 that both dynamics are truly low-dimensional with only one or two modes playing a dominant role, respectively.

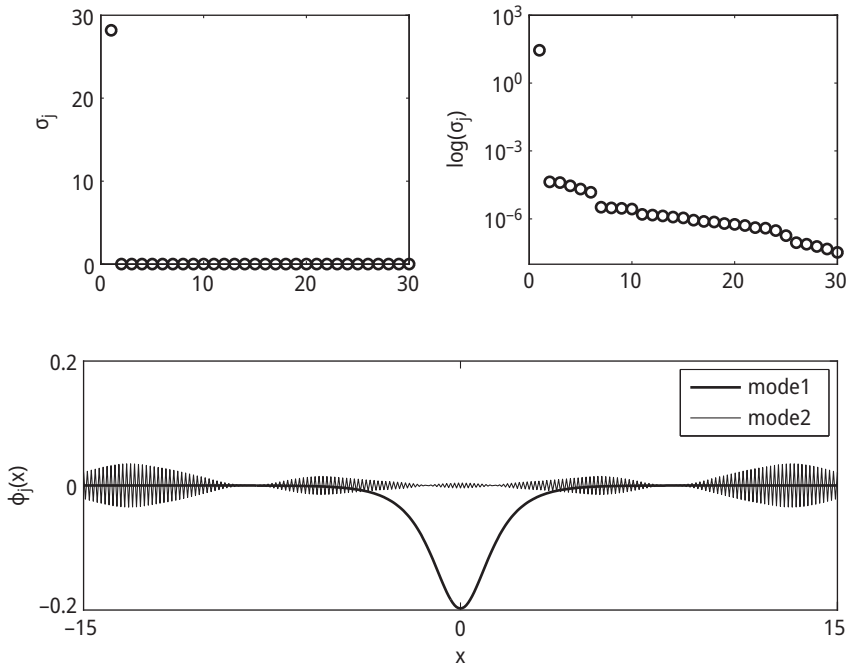


Figure 20.3 Projection of the $N = 1$ evolution to POD modes. The top two figures are the singular values σ_j of the evolution demonstrated in (20.1) on both a regular scale and log scale. This demonstrates that the $N = 1$ soliton dynamics is primarily a single mode dynamics. The first two modes are shown in the bottom panel. Indeed, the second mode is meaningless and is generated from noise and numerical round-off.

20.2.1 $N = 1$ soliton reduction

To take advantage of the low-dimensional structure, we first consider the $N = 1$ soliton dynamics. Figure 20.1 shows that a single mode in the SVD dominates the dynamics. This is the first column of the $\mathbf{U}$ matrix (see Sec. 15.5). Thus, the dynamics is recast in a single mode so that

$$u(x,t) = a(t)\phi(x) \tag{20.2.1}$$

where now there is a sum in (20.1.4) since there is only a single mode $\phi(x)$. Plugging in this equation into the NLS equation (20.2.2) yields the following:

$$ia_t\phi + \frac{1}{2}a\phi_{xx} + |a|^2a|\phi|^2\phi = 0\,. \tag{20.2.2}$$

The inner product is now taken with respect to ϕ, which gives

$$ia_t + \frac{\alpha}{2}a + \beta|a|^2a = 0 \tag{20.2.3}$$

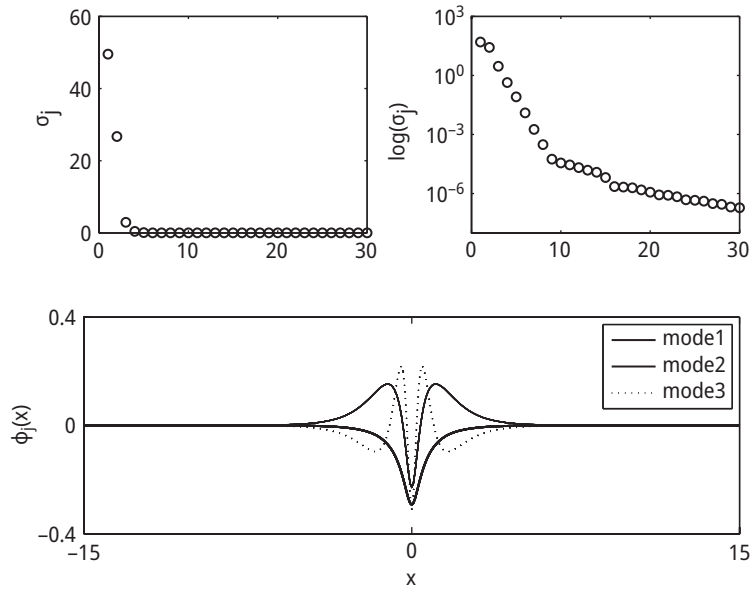


Figure 20.4 Projection of the $N = 2$ evolution to POD modes. The top two figures are the singular values σ_j of the evolution demonstrated in (20.2) on both a regular scale and log scale. This demonstrates that the $N = 2$ soliton dynamics is primarily a two-mode dynamics, as these two modes contain approximately 95% of the evolution energy. The first three modes are shown in the bottom panel.

where

$$\alpha = \frac{(\phi_{xx}, \phi)}{(\phi, \phi)} \tag{20.2.4a}$$

$$\beta = \frac{(|\phi|^2\phi, \phi)}{(\phi, \phi)} \tag{20.2.4b}$$

and the inner product is defined as integration over the computational interval so that $(\phi, \psi) = \int \phi\psi^* dx$. Note that the integration is over the computational domain and the $*$ denotes the complex conjugate.

The differential equation for $a(t)$ given by (20.2.3) can be solved explicitly to yield

$$a(t) = a(0) \exp\left[i\frac{\alpha}{2}t + \beta|a(0)|^2 t\right] \tag{20.2.5}$$

where $a(0)$ is the initial value condition for $a(t)$. To find the initial condition, recall that

$$u(x, 0) = \operatorname{sech}(x) = a(0)\phi(x)\,. \tag{20.2.6}$$

Taking the inner product with respect to $\phi(x)$ gives

$$a(0) = \frac{(\operatorname{sech}(x), \phi)}{(\phi, \phi)}\,. \tag{20.2.7}$$

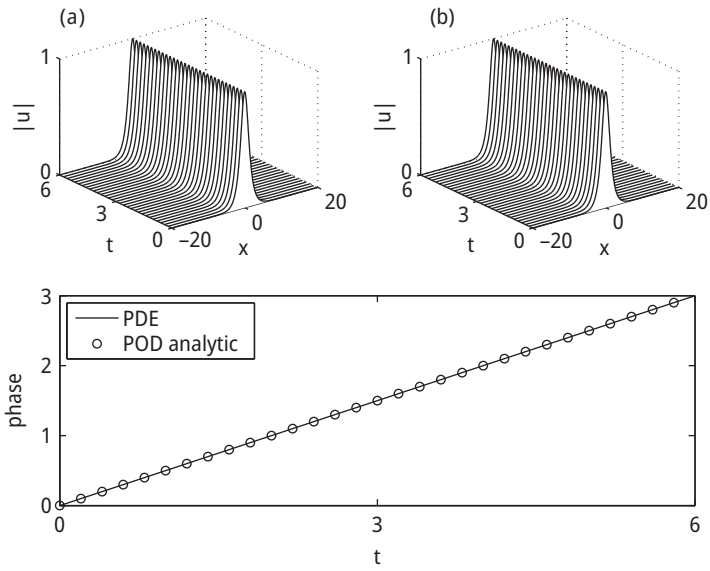


Figure 20.5 Comparison of (a) full PDE dynamics and (b) one-mode POD dynamics. In the bottom figure, the phase of the pulse evolution at $x = 0$ is plotted for the full PDE simulations and the one-mode analytic formula (circles) given in (20.2.8).

Thus, the one-mode expansion gives the approximate PDE solution

$$u(x,t) = a(0)\exp\left[i\frac{\alpha}{2}t + \beta|a(0)|^2 t\right]\phi(x)\,. \tag{20.2.8}$$

This solution is the low-dimensional POD approximation of the PDE expanded in the best basis possible, i.e. the SVD-determined basis.

For the $N = 1$ soliton, the spatial profile remains constant, while its phase undergoes a nonlinear rotation. The POD solution (20.2.8) can be solved exactly to give a characterization of this phase rotation. Figure 20.5 shows both the full PDE dynamics along with its one-mode approximation. This suggest that the $N = 1$ behaviour is indeed a single mode dynamics. This is also known to be true from other solution methods, but it is a critical observation that the POD method for PDEs reproduces all of the expected dynamics.

20.2.2 $N = 2$ soliton reduction

The case of the $N = 2$ soliton is a bit more complicated and interesting. In this case, two modes clearly dominate the behaviour of the system. These two modes are the first two columns of the matrix $\mathbf{U}$ and are now used to approximate the dynamics observed in Fig. 20.2. In this case, the two-mode expansion takes the form

$$u(x,t) = a_1(t)\phi_1(x) + a_2(t)\phi_2(x) \tag{20.2.9}$$

where the ϕ_1 and ϕ_2 are simply taken from the first two columns of the $\mathbf{U}$ matrix in the SVD. Inserting this approximation into the governing equation (20.2.2) gives

$$i\left(a_{1t}\phi_1 + a_{2t}\phi_2\right) + \frac{1}{2}\left(a_1\phi_{1xx} + a_2\phi_{2xx}\right) + \left(a_1\phi_1 + a_2\phi_2\right)^2\left(a_1^*\phi_1^* + a_2^*\phi_2^*\right) = 0. \quad (20.2.10)$$

Multiplying out the cubic term gives the equation

$$\begin{aligned} &i\left(a_{1t}\phi_1 + a_{2t}\phi_2\right) + \frac{1}{2}\left(a_1\phi_{1xx} + a_2\phi_{2xx}\right) \\ &\quad + \left(|a_1|^2a_1|\phi_1|^2\phi_1 + |a_2|^2a_2|\phi_2|^2\phi_2 + 2|a_1|^2a_2|\phi_1|^2\phi_2 + 2|a_2|^2a_1|\phi_2|^2\phi_1\right. \\ &\quad \left. + a_1^2a_2^*\phi_1^2\phi_2^* + a_2^2a_1^*\phi_2^2\phi_1^*\right). \end{aligned} \quad (20.2.11)$$

All that remains is to take the inner product of this equation with respect to both $\phi_1(x)$ and $\phi_2(x)$. Recall that these two modes are orthogonal, thus the resulting 2×2 system of nonlinear equations results

$$\begin{aligned} &ia_{1t} + \alpha_{11}a_1 + \alpha_{12}a_2 + \left(\beta_{111}|a_1|^2 + 2\beta_{211}|a_2|^2\right)a_1 \\ &\quad + \left(\beta_{121}|a_1|^2 + 2\beta_{221}|a_2|^2\right)a_2 + \sigma_{121}a_1^2a_2^* + \sigma_{211}a_2^2a_1^* = 0 \quad (20.2.12b) \\ &ia_{2t} + \alpha_{21}a_1 + \alpha_{22}a_2 + \left(\beta_{112}|a_1|^2 + 2\beta_{212}|a_2|^2\right)a_1 \\ &\quad + \left(\beta_{122}|a_1|^2 + 2\beta_{222}|a_2|^2\right)a_2 + \sigma_{122}a_1^2a_2^* + \sigma_{212}a_2^2a_1^* = 0 \quad (20.2.12a) \end{aligned}$$

where

$$\alpha_{jk} = (\phi_{j_{xx}}, \phi_k)/2 \quad (20.2.13a)$$

$$\beta_{jkl} = \left(|\phi_j|^2\phi_k, \phi_l\right) \quad (20.2.13b)$$

$$\sigma_{jkl} = \left(\phi_j^2\phi_k^*, \phi_l\right) \quad (20.2.13c)$$

and the initial values of the two components are given by

$$a_1(0) = \frac{(2\mathrm{sech}(x), \phi_1)}{(\phi_1, \phi_1)} \quad (20.2.14a)$$

$$a_2(0) = \frac{(2\mathrm{sech}(x), \phi_2)}{(\phi_2, \phi_2)}. \quad (20.2.14b)$$

This gives a complete description of the two-mode dynamics predicted from the SVD analysis.

The 2×2 system (20.2.12a) can be easily simulated with any standard numerical integration algorithm (e.g. fourth-order Runge–Kutta). Before computing the dynamics, the inner products given by α_{jk}, β_{jkl}, and σ_{jkl} must be calculated along with the initial conditions $a_1(0)$ and $a_2(0)$. Upon completion of these calculations, *ode45* can then be used to solve the system (20.2.12a) and extract the dynamics observed in Fig. 20.6. Note that the two-mode

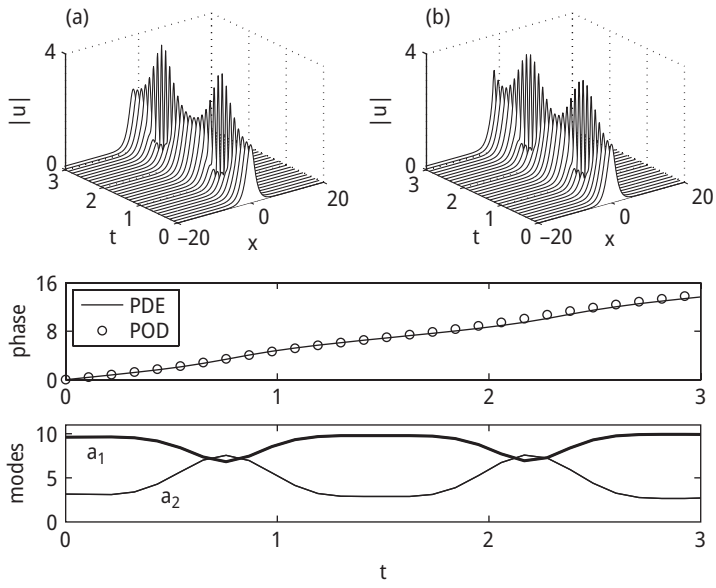


Figure 20.6 Comparison of (a) full PDE dynamics and (b) two-mode POD dynamics. In the bottom two figures, the phase of the pulse evolution at $x = 0$ (middle panel) is plotted for the full PDE simulations and the two-mode dynamics along with the nonlinear oscillation dynamics of $a_1(t)$ and $a_2(t)$ (bottom panel).

dynamics does a good job in approximating the solution. However, there is a phase drift that occurs in the dynamics that would require higher precision in both taking time slices of the full PDE and more accurate integration of the inner products for the coefficients. Indeed, the simplest trapezoidal rule has been used to compute the inner products and its accuracy is somewhat suspect. Higher-order schemes could certainly help improve the accuracy. Additionally, incorporation of the third mode could also help. In either case, this demonstrates sufficiently how one would in practice use the low-dimensional structures for approximating PDE dynamics.

20.3 The POD Method and Symmetries/Invariances

The POD reduction technique can be a powerful tool in projecting seemingly complex dynamics to a low-dimensional manifold where all the *important* dynamics occurs. Of course, it is not without its drawbacks and limitations. In particular, the method can often fail if the data sampled is simply put through the SVD algorithm without considering its basic structure and potential invariances. Invariances can arise due to various symmetry considerations in a given physical system, for instance, translational or rotational invariance.

As an example, we can once again consider the N-soliton dynamics as illustrated in Figs. 20.1–20.4. In this example, the N-soliton ($N = 1$ and $N = 2$) solution of the NLS equation was integrated using a spectral method in order to generate the data for sampling. A small

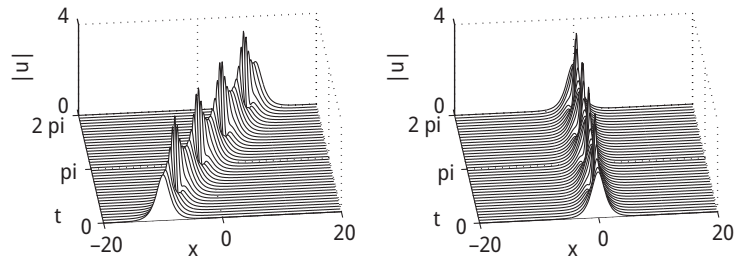


Figure 20.7 Evolution dynamics of the $N = 2$ soliton solution when subject to a centre-frequency shift (left panel) and when the translational dynamics is factored out (right panel). The translating 2-soliton generates a high-dimensional singular value distribution. In contrast, by factoring out the translational mode, the distribution once again becomes dominated by two modes as previously shown in Figs. 20.1–20.4.

change to this simulation can make a tremendous difference in the results of the POD reduction scheme. Specifically, consider integrating once again the NLS equation (20.2.2) but with the initial condition

$$u(x, 0) = N\text{sech}(x + x_0)\exp(i\Omega x) \tag{20.3.1}$$

where Ω represents a centre-frequency shift of the soliton and x_0 is a centre-position offset variable. The centre-frequency perturbation to the N-soliton solution causes the soliton solution to move at a prescribed group velocity. Indeed, such a group-velocity movement of the pulse, when the centre-frequency is driven by noise, leads to a fundamental limit in fibre optical communication systems known as the Gordon–Haus jitter [152, 153].

To observe the impact of the centre-frequency on the soliton propagation, consider numerically integrating the NLS equation with the above initial conditions and $N = 2, \Omega = \pi$, and $x_0 = 10$. Figure 20.7 (left panel) shows the soliton evolution. It is clearly seen that the centre-frequency causes a group-velocity drift in the pulse. Other than that, however, the $N = 2$ soliton dynamics appears to be identical to what is observed and characterized in Figs. 20.1–20.4. As before, we could arrange our numerical simulations into a data matrix and perform an SVD reduction. The bottom left panel of Fig. 20.8 shows the singular values that would result from such a reduction. It is clear in this case that such a method would not result in a reduction of the dynamics as the translating data, when correlated across time and space, do not produce redundant data. Indeed, there is a very slow decay of the singular values and the POD method cannot be used to generate a low-dimensional manifold for the dynamics.

The failure of the method in this case is due simply to the translational invariance. If the invariance is *removed*, or factored out [154], before a data reduction is attempted, then the POD method can once again be used. In order to remove the invariance, the invariance

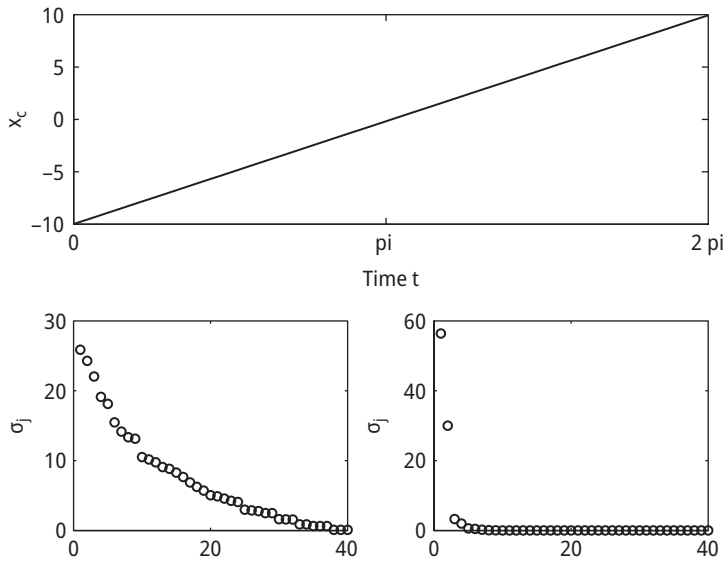


Figure 20.8 Dynamics of the centre position $x_c(t)$ (top panel) showing the constant translation of the 2-soliton to the right in the computational domain. The distribution of singular values for the two data sets demonstrated in Fig. 20.7 are exhibited in the bottom panels. Without factoring out the translation, the SVD does not produce low-dimensional behaviour (bottom left), whereas appropriate factoring of the data reveals the dominant two-mode dynamics (bottom right).

must first be identified and an auxiliary variable defined. Thus, we consider the dynamics rewritten as

$$u(x,t) \to u(x - c(t)) \tag{20.3.2}$$

where $c(t)$ corresponds to the translational invariance in the system responsible for limiting the POD method. The parameter c can be found by a number of methods. Rowley and Marsden [154] propose a template-based technique for factoring out the invariance. Here, a simple centre-of-mass calculation will be used to compute the location of the soliton.

The following code can be implemented to find the centre position of the pulse as it evolves in time:

```
c = np.zeros(len(t))
for j in range(len(t)):
    com = x * np.abs(usol[j, :])**2
    com2 = np.abs(usol[j, :])**2
    c[j] = np.trapz(com, x) / np.trapz(com2, x)
```

The end result of this computation is the production of the vector or function $\mathbf{c}$, which prescribes the location of the pulse. The centre position can then be factored out by the following lines of code:

```
usol_shift = np.zeros_like(usol)
for j in range(len(t)):
    mn, jj = np.min(np.abs(x - c[j])), np.argmin (np.abs(x - c[j]))
        ns = n // 2 - jj
    usift = np.concatenate((usol[j, :], usol[j, :],
         usol[j, :]))
    usol_shift[j, :] = usift[n  - ns:2*n - ns]
```

The new matrix **usol_shift** has now had the group-velocity drift factored out. Figure 20.7 (right panel) shows the net result of this. The centre position vector as a function of time is demonstrated in the top panel of Fig. 20.8. Once the translation is factored out, the SVD reduction of the remaining data set produces the singular value in the bottom right panel of Fig. 20.8. This singular value distribution is identical to what was previously demonstrated in the simple $N = 2$ case. Thus, the dynamics is now three-dimensional: two modes interact nonlinearly to describe the pulse changes, while one mode describes the translation. This example serves to demonstrate that if the invariance is handled properly, only an extra single degree of freedom is required to account for its action in the dynamics.

20.3.1 Advection–Diffusion equations

A more sophisticated example of handling such an invariance is provided by the advection–diffusion equations (see Sec. 9.3):

$$\frac{\partial \omega}{\partial t} + [\psi, \omega] = \nu \nabla^2 \omega \tag{20.3.3a}$$

$$\nabla^2 \psi = \omega \tag{20.3.3b}$$

where

$$[\psi, \omega] = \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} \tag{20.3.4}$$

and $\nabla^2 = \partial_x^2 + \partial_y^2$ is the two-dimensional Laplacian. A spectral method has been developed for solving this problem in Sec. 9.3. Here, the initial data applied to this problem will be a dipole-type initial condition

$$\omega(x, y, 0) = \exp[-(x+8)^2 - (y-2)^2] - \exp[-(x+8)^2 - (y+2)^2]\,. \tag{20.3.5}$$

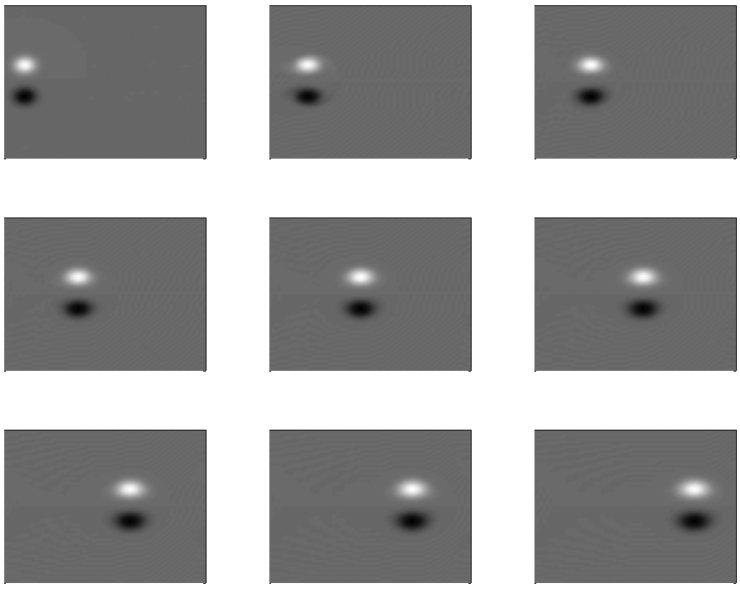


Figure 20.9 Evolution dynamics of a dipole-like initial condition in evenly spaced increments over the time frame $t \in [0, 135]$. The dipole is seen to generate a translation of the structure to the right of the computational domain.

Such an initial condition leads to the formation of a dipole pair that translates to the right as depicted in Fig. 20.9. A nearly constant velocity is displayed as the dipole pair is advected to the right of the computational domain.

As previously noted, the dimensionality of the dynamics can be assessed by using the SVD on the computational data. In this case, the solution is first allowed to settle to the dipole state before sampling is applied. For this case, only the last 40 of 60 slices of this data will be used in our POD projection. Given that the data are two-dimensional, they must first be reshaped into vector form before applying the SVD.

The top left panel of Fig. 20.10 shows the distribution of singular values after sampling the dynamics. Again, without taking care of the translation, the dynamics appears to be high-dimensional. However, it is clear from Fig. 20.9 that the dynamics should indeed be low-dimensional with a dominant dipole mode translating from left to right.

As previously, a computation must be made of the translation of the dipole mode. Here, since the data are symmetric about $y = 0$, it only remains to find the centre-of-mass via a standard moment method. The resulting translation vector is plotted in the bottom panel of Fig. 20.10. This gives the auxiliary variable that is to be factored out of the dynamics. The top right panel of Fig. 20.10 shows the singular value distribution for this factored case. Note that the dynamics, as expected, is dominated by a single mode, i.e. the dipole mode. Figure 20.11 shows the first four modes of the SVD reduction. The first mode is clearly the dominant dipole solution. Modes two through four show the effects of the advection at the

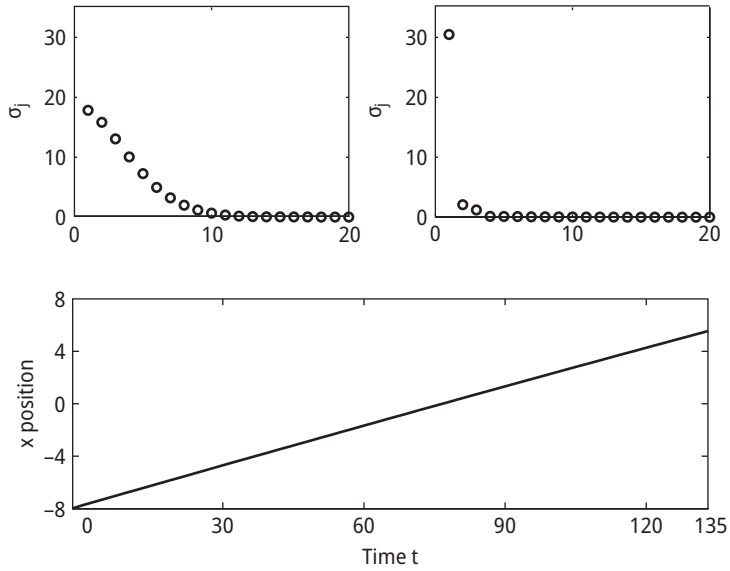


Figure 20.10 Dynamics of the centre position (bottom panel) showing the constant translation of the dipole solution to the right in the computational domain. The distribution of singular values for the raw data and factored out data are exhibited in the top panels. Without factoring out the translation, the SVD does not produce low-dimensional behaviour (top left), whereas appropriate factoring of the data reveals a dominant one-mode dynamic (top right).

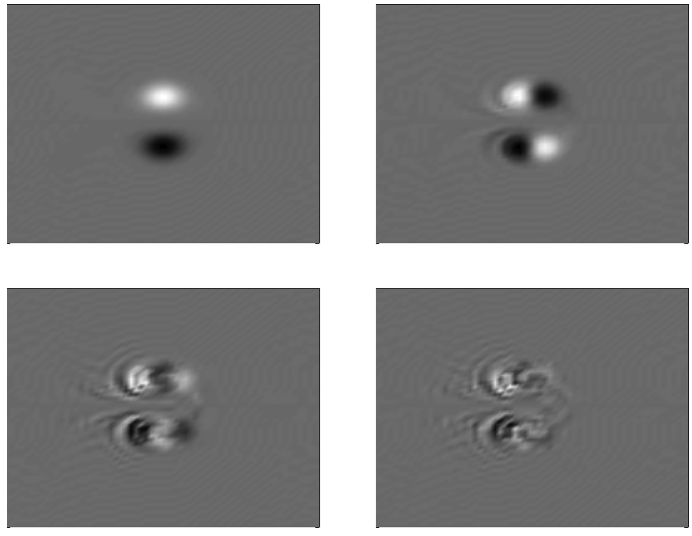


Figure 20.11 The four most dominant modes of the SVD decomposition for the data that has factored out the translational invariance. The first mode (top right) contains 88% of the modal energy, while the first four contains 98%.

back of the dipole, i.e. fluid is ejected out the back of the dipole as it moves towards the left. Note that 88% of the energy is in the first dipole mode, while 98% is in the first four modes.

A one-mode decomposition of the data can be performed by assuming a solution of the form

$$\omega(x,y,t) = a(t)\phi(x,y) \tag{20.3.6}$$

where $\phi(x,y)$ is the dominant dipole mode. Once the vorticity is determined, the streamfunction can be computed from

$$\nabla^2\psi = \omega = a(t)\phi(x,y) \quad \rightarrow \quad \psi = a(t)L^{-1}\phi(x,y) \tag{20.3.7}$$

where the operator L^{-1} inverts the Laplacian operator. Plugging the one-mode expansion into the governing equations and taking the inner product of both sides of the equation for vorticity yields

$$\frac{da}{dt} = \alpha a + \beta a^2 \tag{20.3.8}$$

where

$$\alpha = \nu\frac{(\nabla^2\phi,\phi)}{(\phi,\phi)} \tag{20.3.9a}$$

$$\beta = \frac{((L^-1\phi)_y\phi_x,\phi)}{(\phi,\phi)} - \frac{((L^-1\phi)_x\phi_y,\phi)}{(\phi,\phi)} \tag{20.3.9b}$$

where the coefficients α and β are determined from inner products once the mode structures are computed from the SVD. Due to symmetry considerations of the dipole solution, $\beta = 0$, thus reducing everything to the calculation of α alone. This yields a value of $\alpha = -1.771\nu$. Thus, the leading order (dominant) POD solution for the dipole is simply exponential decay given by

$$a(t) = a(0)\exp(-1.771\nu t)\,, \tag{20.3.10}$$

where $a(0) = (\omega(x,y,0),\phi(x,y))$. In total then, the overall solution has been deconstructed into its constitute parts. To reconstruct, one simply needs to put the three pieces together: the time dynamics of the mode amplitude, the dominant mode structure, and the translational dynamics. This gives the one-mode expansion for the dipole dynamics:

$$\omega(x,y,t) = a(0)\exp(-1.771\nu t)\phi(x-c(t),y)\,. \tag{20.3.11}$$

Of course, a more complicated dynamics would result if a higher-order mode expansion was assumed. However, 88% of the energy is in this single mode, which was handled nicely by factoring out the translation.

To conclude this discussion, one can easily imagine factoring out a myriad of other symmetries. The most obvious is translation, but perhaps the second most obvious is factoring out a rotation. This might arise, for example, when considering spiral wave dynamics in reaction–diffusion systems. Ultimately, this is simply a data preprocessing step, which must be performed prior to the dimensionality reduction step.

20.4 POD Using Robust PCA

The POD technique is ideal for sampling data, determining the low-dimensional structure therein, and exploiting the low-rank structure in the context of dynamical systems. The examples considered to this point demonstrate this by using the POD basis in combination with Galerkin projection in order to reduce the dynamics of a given complex system to a low-dimensional manifold on which the dynamics occurs. At the heart of this dimension reduction is the SVD and its various guises, most notably PCA or POD. The SVD produces characteristic features (principal components) that are determined by the covariance matrix of the data. Fundamental in producing the SVD/PCA/POD modes is the L^2 norm for data fitting. Although the L^2 norm is highly appealing due to its centrality (and physical interpretation) in most applications as an *energy*, it does have potential flaws that can severely limit its applicability when trying to integrate it with dynamical systems theory. In particular, if the data gathered through measurement is corrupt or suffers from large and sparse noise fluctuations, the higher-dimensional least-square fitting to the data matrix by the SVD algorithm squares this pernicious error; leading to significant deformation of the singular values and PCA/POD modes describing the data. Although many physical systems and/or computations are relatively free of data corruption, many modern applications, such as PIV fluid measurements, are rife with arbitrary corruption of the measured data due to the sensitivity of measuring *derivative*-type data. This ensures that the standard application of the SVD will be of limited use.

Ideally, in performing dimensionality reduction, one should not allow the corrupt or large noise fluctuations to so strongly influence one's results. In order to limit the impact of such *data outliers*, a different measure, or norm, can be envisioned in measuring the best data fit (SVD/PCA/POD modes). One measure that has recently produced great success in this arena is the L^1 norm [55], which no longer squares the distance of the data to the best-fit mode. Indeed, the L^1 norm has been demonstrated to promote sparsity and is the underlying theoretical construct in *compressive sensing* or *sparse sensing* algorithms (see Sec. 14.10 for further details). Here, the L^1 norm is simply used as an alternative measure (norm) for performing the equivalent of the least-square fit to the data. As a result, a more robust method is achieved for computing PCA components, i.e. the so-called *robust PCA* method.

The robust PCA technique has already been considered in Sec. 15.6. Briefly, the idea of the robust PCA is to provide an algorithm capable of separating in an efficient manner low-rank data with corrupt (sparse) measurement/matrix error. Thus, the idea would be to consider a matrix, which is composed of these two elements together

$$\mathbf{M} = \mathbf{L} + \mathbf{S} \tag{20.4.1}$$

where $\mathbf{M}$ is the data matrix of interest that is composed of a low-rank structure $\mathbf{L}$ along with some sparse data $\mathbf{S}$. The data-analysis objective is the following: given measurements or observations $\mathbf{M}$, can the low-rank matrix $\mathbf{L}$ and sparse matrix $\mathbf{S}$ be recovered? Adding to the difficulty of this task is the following: neither the rank of the matrix $\mathbf{L}$, nor the number of nonzero (sparse) elements of the matrix $\mathbf{S}$ are known.

The separation problem seems impossible to solve since the number of unknowns to infer for $\mathbf{L}$ and $\mathbf{S}$ is twice as many as the given measurements in $\mathbf{M}$. However, Candés and co-workers [55] have recently proved that this can indeed be solved through the formulation of a tractable convex optimization problem. In real applications, however, it is rare that the matrix decomposition is as ideal as (20.4.1). More generally, the decomposition is of the form

$$\mathbf{M} = \mathbf{L} + \mathbf{S} + \mathbf{N} \tag{20.4.2}$$

where the matrix $\mathbf{N}$ is a dense, small perturbation accounting for the fact that the low-rank component is only approximately low rank and that small errors can be added to all the entries. But even in this case, the convex optimization algorithm for generating robust PCA seems to do a remarkable job at separating low-rank from sparse data. This can be used to great advantage when certain types of data are considered, namely those with corrupt data or large, sparse noisy perturbations.

20.4.1 POD mode selection via robust PCA

To illustrate the role of robust PCA in dimensionality reduction of dynamical systems, we once again revisit the two-soliton example considered throughout this chapter. The two-soliton dynamics are illustrated in Figs. 20.1–20.4. It is once again demonstrated in Fig. 20.12 along with a modified two-soliton data matrix, which has been corrupted by sparse noise fluctuations. Assuming the two-soliton (NLS solver) has been executed with the initial condition $u(x,0) = 2\,\mathrm{sech}\,x$ and over the interval $t \in [0, 2\pi]$, then the resulting complex solution is contained in the matrix **usol**. This matrix is corrupted by 60 randomly chosen, sparse noise

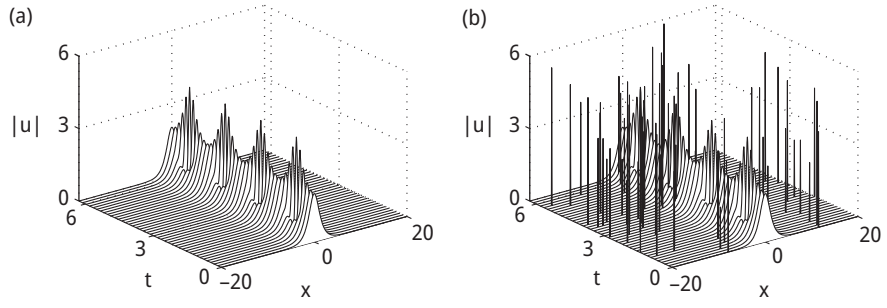


Figure 20.12 (a) Ideal evolution dynamics of the two-soliton of the nonlinear Schrödinger equation, and (b) the ideal data matrix corrupted by a sparse set of 60 measurements.

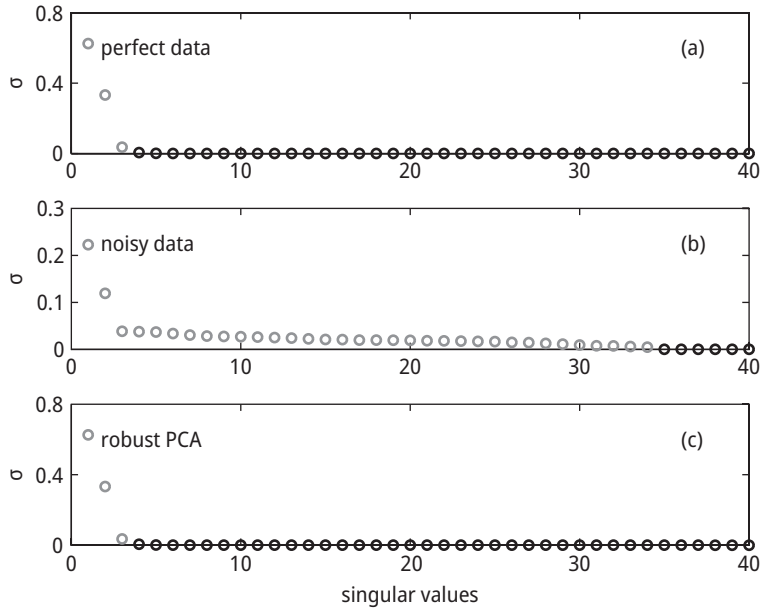


Figure 20.13 Singular values of the data matrices for the ideal data (top panel), the corrupted data (middle panel), and the low-rank data computed through robust PCA (bottom panel). The robust PCA construction produces almost a perfect match to the original, ideal data. The magenta dots represent the number of modes necessary to construct 99% of the original data. In this case, the ideal and robust PCA require three modes while the corrupt data requires 34 modes.

fluctuations. Recall that the noise added has both real and imaginary parts as is the case for the complex evolution field given by the NLS equation.

Previously, the data matrix was used in its raw (and perfect) form to generate the low-dimensional manifold spanned by a reduced set of POD modes. However, with the sparse corruption of the data, it seems unlikely that the desired low-dimensional manifold can be computed. Application of the SVD to the ideal data and corrupt data shows significant differences. Figures 20.13 and 20.14 show the singular values and POD modes for both reductions. In the top panel of both figures are the results for the ideal data, showing that a clear two- to three-mode dominance exists in the data, i.e. three modes (magenta dots) captures more than 99% of the data matrix. Indeed, this was directly exploited in Sec. 20.2 for the construction of a low-dimensional manifold for the dynamical evolution of the system. In contrast, the middle panels show that the corrupted data matrix activates a significant number of modes. Specifically, it now takes 34 modes (magenta circles) in order to capture 99% of the corrupt data matrix. The POD modes are also now severely misshaped from their ideal, noisy free state.

To remove the corrupted data, a sparse and low-rank decomposition is made. Usually, there is a parameter $lambda$ that can be adjusted to best separate the low-rank from sparse data. The real and imaginary portions are put back together at the end of the algorithm

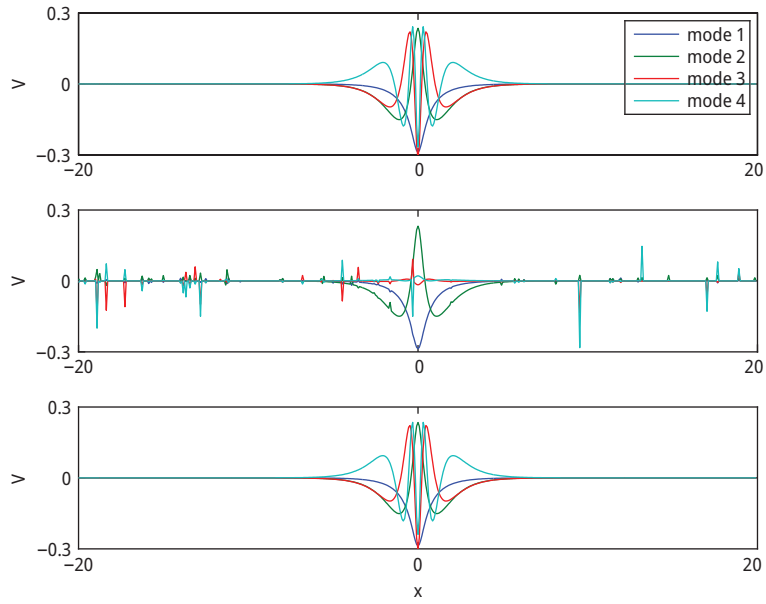


Figure 20.14 Dominant modes for the ideal data (top panel), the corrupted data (middle panel), and the low-rank data computed through robust PCA (bottom panel). The ideal and robust PCA produce the same dominant two- to three-modes, while the corrupt data produces highly erratic modes.

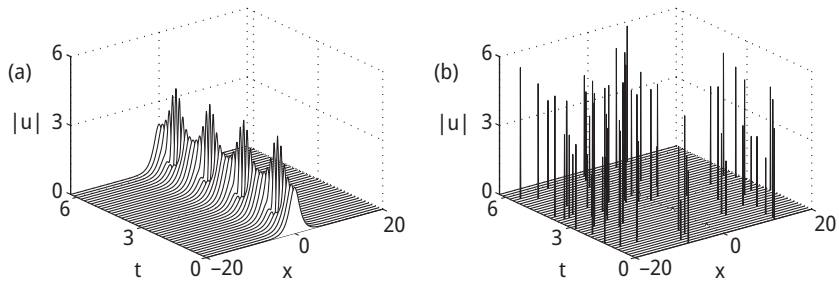


Figure 20.15 Separation of the original corrupt data matrix shown in Fig. 20.12(b) into a low-rank component (a) and a sparse component (b). The separation is almost perfect, with the low-rank matrix being dominated by two- to three-modes, i.e. the three modes contain greater than 99% of the energy.

in order to SVD the low-rank portion of interest, i.e. the portion of the data considered to be without corrupt data. The two matrices corresponding to the low-rank and sparse portions of the data matrix are plotted in Fig. 20.15. The parameter $lambda$ used in the **inexact_alm_rpca** algorithm can be tuned to best separate the sparse from low-rank matrices in (1). Here, a value of 0.2 is used, which produces an almost ideal separation as is shown in Fig. 20.13 (bottom panel), Fig. 20.14 (bottom panel), and Fig. 20.15. Indeed, an almost perfect separation is achieved as advertised (guaranteed) by Candés and co-workers [55].

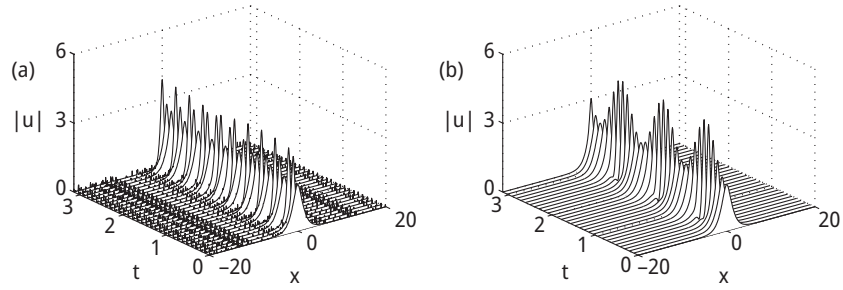


Figure 20.16 Comparison of the two-mode POD evolution given by (20.2.12a) of Sec. 20.2 using the (a) SVD reduction of the corrupt data matrix and (b) using the low-rank data generated from the robust PCA algorithm. The use of the robust PCA algorithm renders the POD results almost identical to those with the ideal data, thus suggesting it can potentially be a filter for corrupt data before applying POD reductions.

Finally, the POD dynamics are compared for a two-mode truncation of the corrupt data versus the low-rank approximation of the data achieved through robust PCA. Figure 20.16 shows the results of both two-mode expansions. The robust PCA modes produce almost identical results to those achieved with the standard POD reduction using the ideal data. In contrast, the POD reduction using the corrupt data produces inner products in (20.2.12a) of Sec. 20.2 that greatly skew their value and their ability to predict accurately the true underlying evolution. Indeed, the two POD mode dynamics is significantly deteriorated from its robust PCA counterpart.

As was already pointed out in Sec. 15.6, the robust PCA algorithm advocated for here is quite remarkable in its ability to separate sparse corruption from low-rank structure in a given data matrix. The application of robust PCA essentially acts as an advanced filter for cleaning up a data matrix. Indeed, one can potentially use this as a filter, which is very unlike the time–frequency filtering schemes considered previously. This has the potential to greatly enhance the POD reduction schemes advocated here when they are subject to corrupt data measurements.

20.5 Shallow Recurrent Decoders (SHRED) for Sensing and PDEs

Already covered in Ch. 15 as a generalization of the SVD, the SHRED architecture is revisited here in the context of PDEs and spatio-temporal data [124]. The method's theoretical foundations are also once again considered from the point of view of separation of variables and the Takens' embedding theorem. Indeed, the SHRED architecture is based upon the separation of variables technique for solving linear PDEs [177]. Separation of variables assumes that a solution can be separated into a product of spatial and temporal functions $u(x,t) = T(t)X(x)$. This solution form is then used to reduce the PDE into ordinary

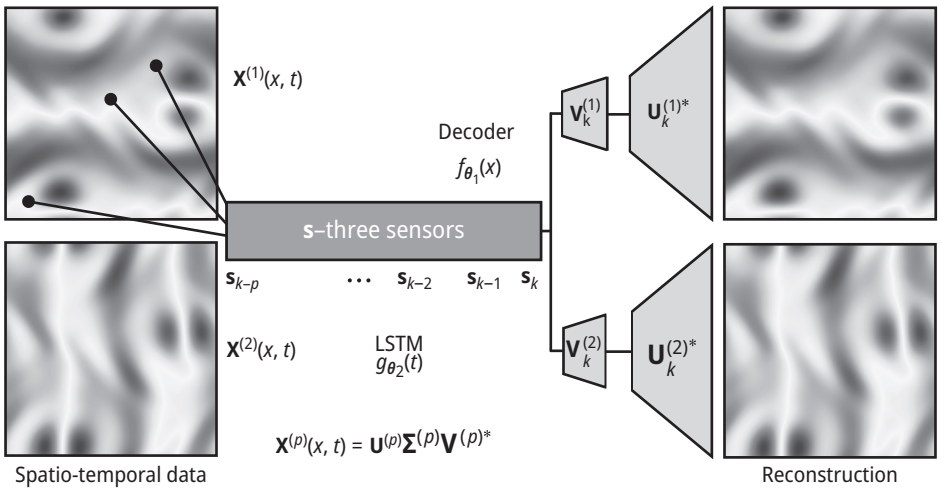


Figure 20.17 SHRED architecture for compressive training. The spatio-temporal data is first compressed with the SVD so that the training learns a mapping from sensor measurements $\mathbf{s}$ to the temporal evolution in the compressed space $\mathbf{V}$. The sensors from a single field can be used to map to all other fields provided the dynamics are coupled. This allows for laptop-level training.

differential equations: one for time $T(t)$ and one for space $X(x)$. Such a decomposition also constitutes the underpinnings of spectral methods for the numerical solution of the PDE, linear or nonlinear. What is demonstrated is the method trained on the architecture of Fig. 20.17. Specifically, SHRED is trained in a compressive space allowing for as little as three sensors on a single field to reconstruct all other fields, even those unmeasured, accurately.

Consider the constant coefficient linear PDE

$$\dot{u} = \mathcal{L}(\partial_x, \partial_x^2, \cdots)u \tag{20.5.1}$$

where $u(x,t)$ specifies the spatio-temporal field of interest. Typically, the initial condition (IC) and boundary conditions (BCs) are given by

$$\text{IC:} \qquad u(x,0) = u_0(x) \tag{20.5.2a}$$

$$\text{BCs:} \qquad \alpha_1 u(0,t) + \beta_1 u_x(0,t) = g_1(t) \text{ and} \tag{20.5.2b}$$

$$\alpha_2 u(L,t) + \beta_2 u_x(L,t) = g_2(t). \tag{20.5.2a}$$

This may be generalized to systems of several spatial variables, or a system with no time dependence. The linear operator $\mathcal{L}$ specifies the spatial derivatives, which in turn model the underlying physics of the system. Simple examples of $\mathcal{L}$ include $\mathcal{L} = c\partial_x$ (the one-way wave equation) and $\mathcal{L} = \kappa\partial_x^2$ (the heat equation).

The earliest solutions of linear PDEs assumed separation of variables, whereby $u(x,t) = \exp(\lambda t)X(x)$ was a product of a temporal (exponential) function multiplied by a

spatial function. The parameter λ is in general complex. This gives the eigenfunction solution of (20.5.1) to be

$$u(x,t) = \sum_{n=1}^{N} a_n \exp(\lambda_n t)\phi_n(x) \tag{20.5.3}$$

where $\phi_n(x)$ are the eigenfunctions of the linear operator and λ_n are its eigenvalues ($\mathcal{L}\phi_n(x) = \lambda_n\phi_n(x)$). Here, a finite-dimensional approximation N is assumed, which is standard in practice for numerical evaluation.

Typically, initial conditions $u(x,0) = u_0(x)$ are imposed in order to uniquely determine the coefficients b_n. Specifically, at time $t = 0$ (5) becomes

$$u_0(x) = \sum_{n=1}^{N} a_n\phi_n(x). \tag{20.5.4}$$

Taking the inner product of both sides with respect to $\phi_m(x)$ and making use of orthogonality gives

$$a_n = \langle u_0(x), \phi_n(x)\rangle.$$

SHRED instead has measurements at a single spatial (sensor) location x_s, but with a temporal history. Thus if SHRED has, for example, N temporal trajectory points, this gives at each time point of the measurement:

$$u(x_s,t_j) = \sum_{n=1}^{N} a_n \exp(\lambda_n t_j)\phi_n(x_s) \quad \text{for } j = 1,2,\cdots,N.$$

This results in N equations for the N unknowns a_n. Specifically, the $N \times N$ system of equations $\mathbf{Ax} = \mathbf{b}$ is prescribed by the vector components $x_k = a_k$ and $b_k = u(x_s,t_k)$, and matrix components $(a_{kj}) = \exp(\lambda_k t_j)\phi_k(x_s)$. As with the initial condition (20.5.2a), the time trajectory of measurements at a single location uniquely prescribes the solution. This analysis can easily be generalized to include multiple sensor measurements at a single time point. Thus, if there are two measurements at a given time t_j, then only $N/2$ trajectory points are needed to uniquely determine the solution. Likewise, three sensor measurements at a given time requires only $N/3$ trajectory points. In addition to stationary sensor measurements, one can also consider mobile sensors, whereby the measurement of the system is at different locations over time: $x_s = x_{s(t_j)}$. The above arguments are easily modified so that the vector component $b_k = u(x_{s(t_j)},t_k)$ and matrix components $(a_{kj}) = \exp(\lambda_k t_j)\phi_k(x_{s(t_j)})$.

Thus, temporal trajectory information at a single spatial location, or with a moving sensor, is equivalent to knowing the full spatial field at a given point in time. SHRED is a generalization of the separation of variables architecture $u(x,t) = T(t)X(x)$, where the LSTM models time $T(t)$ and the decoder models space $X(x)$. Of course, as a generalization to nonlinear dynamics, rigorous theoretical bounds of SHRED are difficult to achieve, much like analytic and numerical solutions are difficult to rigorously bound in computational PDE settings. But

certainly in the linear limit, the above arguments show explicitly why SHRED is guaranteed to work and recover the full spatio-temporal field exactly.

We can also consider coupled, constant coefficient linear PDEs. For example, the coupled system

$$\dot{u} = \mathcal{L}_1 u + \mathcal{L}_2 v \tag{20.5.5a}$$

$$\dot{v} = \mathcal{L}_3 u + \mathcal{L}_4 v \tag{20.5.5b}$$

where $u(x,t)$ and $v(x,t)$ specifies the spatio-temporal fields of interest. The PDEs can be instead be written in the form

$$\ddot{u} = \mathcal{L}_1 \dot{u} + \mathcal{L}_2 \mathcal{L}_3 u + \mathcal{L}_2 \mathcal{L}_4 \left(\mathcal{L}_2^{-1} (\dot{u} - \mathcal{L}_1 u) \right) \tag{20.5.6}$$

where (20.5.5a) is differentiated with respect to time and (20.5.5b) is used in order to write the PDEs as a function of $u(x,t)$ alone. Thus, knowledge of the field $u(x,t)$ alone is capable of constructing the solution fields $u(x,t)$ and $v(x,t)$. For this second-order (in time) PDE, both an initial condition $u(x,0)$ and an initial *velocity* require specification $\dot{u}(x,0)$ in order to uniquely determine the solution. As with the previous arguments, a time trajectory embedding of $2N$ measurements can be used to uniquely determine the solution.

To show the performance of the method, we consider two-dimensional Kolmogorov flow. Thus, the solver provides numerical solutions to the divergence-free Navier–Stokes equations [178]:

$$\nabla \cdot u = 0 \tag{20.5.7a}$$

$$\partial_t u + u \cdot \nabla u = -\nabla p + \nu \Delta u + f \tag{20.5.7b}$$

where $u(x,y,t)$ is a two-dimensional flow dynamics. The output of the solution code is the velocity field of the flow in both the x- and y-directions.

SHRED can be trained on the compressive space of the flow physics illustrated in Fig. 20.18. Importantly, a compressive space is achieved by using the randomized SVD algorithm

```
from sklearn.utils.extmath import randomized_svd
u,s,v=randomized_svd(X, n_components = 50, n_iter='auto')
```

Both the x-component and y-component of the flow physics are compressed using the SVD. The training is now on the matrix $\mathbf{V}$ of the x- and y-directions of velocity. Once trained, one can sense on a single field alone and reconstruct both velocity fields [155]. Figure 20.20 shows the approximation capabilities of the SHRED architecture against the ground truth data. As can be seen, despite the turbulent nature of the data, the SHRED architecture captures the dynamics quite well. Like most neural network architectures, SHRED does loose

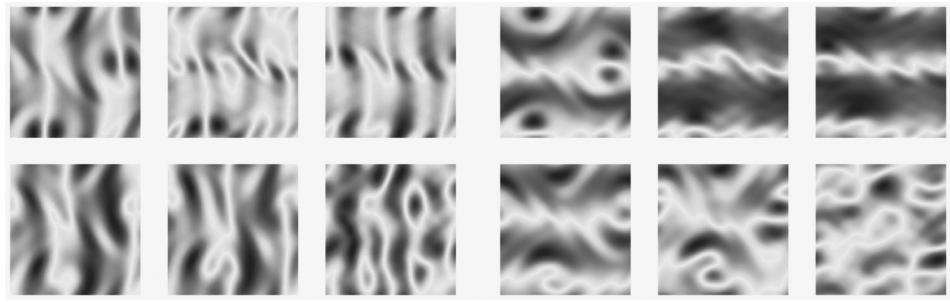


Figure 20.18 Evolution dynamics of the two-dimensional divergence-free Navier–Stokes equations. The left-side panels represent the evolution dynamics of the horizontal components (x-direction) of the velocity with snapshots separated by 500 time steps. The right panels represent the corresponding evolution dynamics of the vertical components (y-direction) of the velocity. The 2D Kolmogorov flow is a prototypical example of turbulent flows. The SHRED architecture trains on three random point sensor locations of a single field, allowing for a compressive training framework capable of reconstruction of the full state-space in both fields.

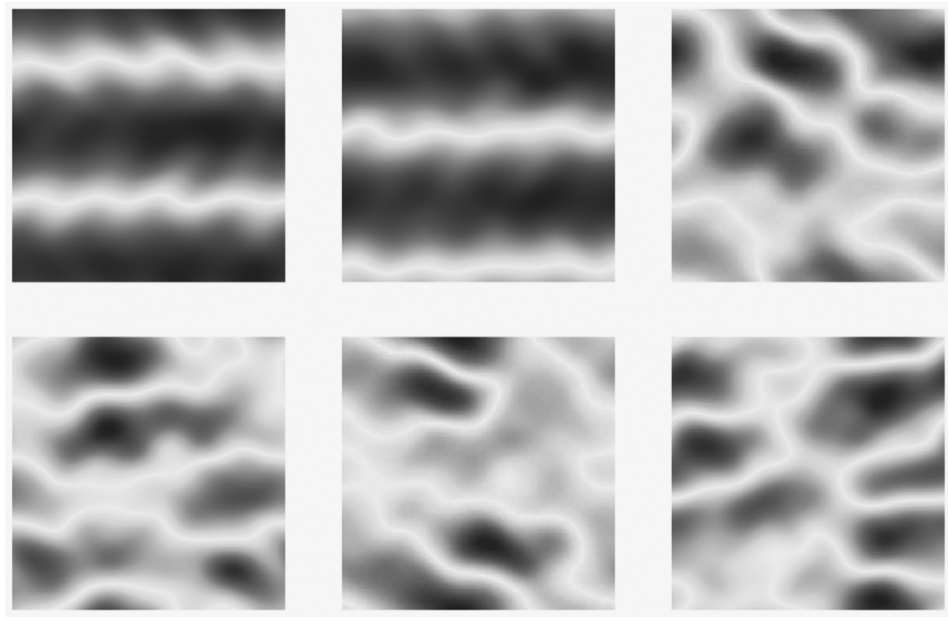


Figure 20.19 The first six SVD modes of evolution dynamics of the vertical components (y-direction) of the velocity in the 2D Kolmogorov flow. A total of 50 modes are retained for training. This allows for the SHRED architecture to train to 50 modes of $\mathbf{V}$ instead of the original state-space, which is of size $18^2 = 16,384$. This compressive training method massively reduces the training time and computational demands required for mapping the sensors to the full state-space.

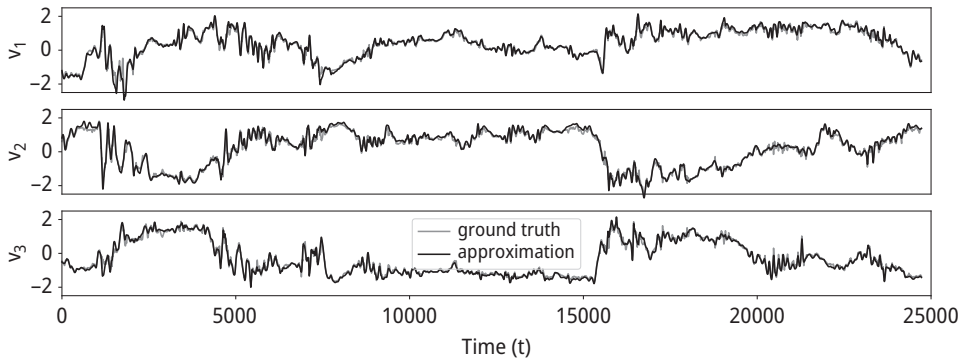


Figure 20.20 Evolution of the first three SVD modes of the matrix $\mathbf{V}$. The magenta lines represent the ground truth $\mathbf{v}_k$ for $k = 1, 2, 3$ of the original training data. The black lines represent the trained reconstruction of the SHRED model. The quality of the $\mathbf{V}$ reconstruction determines the modelling capabilities of the SHRED model, which is quite accurate for reconstruction. Forecasting for such turbulent flows is difficult given the rapid decorrelation time of the dynamics.

information to higher frequencies due to spectral bias [156]. However, the spectral bias is greatly diminished by training on the compressive space as many of the key high-frequency features in space are encoded directly into the SVD projection matrix $\mathbf{U}$. Mobile sensors can also be incorporated into the SHRED architecture, allowing for broader applications [125, 157]. SHRED ultimately gives a robust and powerful framework for modelling spatio-temporal systems with an easy-to-train architecture that is robust, computationally efficient, and requires minimal hyper-parameter tuning.

20.6 Sparse Identification of Nonlinear Dynamics (SINDy)

Physics-based systems in science and engineering are typically modelled by governing equations, with simulations allowing for the exploration of the system for various parameter regimes. This is why computational methods and scientific computing, considered previously in this text, are now an integral part of every field of application. Computation allows for the characterization of many modern high-dimensional, complex dynamical systems, thus enabling engineering design and control in a diversity of application areas. The ubiquity of computing has led to the consideration of *reduced order models* (ROMs) for improving computational efficiency [158, 179, 180, 181]. However, data-driven methods have now emerged as an alternative paradigm, where building models directly from data is advocated. Indeed, in many emerging modern applications, including in turbulence closure modelling, weather forecasting, powergrid networks, climate modelling, and neuroscience, for instance,

the construction of dynamic models from data is necessary, as first principle models are unknown, inadequate or beyond first principles derivations. In what follows, we consider a leading strategy that helps represent physics-based systems by learning dynamics and embeddings jointly.

The data-driven discovery process begins with data acquisition. Thus, sensors are critical for empowering the physics learning process. Often what is measured by the sensors $\mathbf{y}$ is not the correct variable $\mathbf{U}$ for building a parsimonious, or dynamic, model representation. Thus, it is important to first learn how to map the measurements $\mathbf{y}$ to a state-space representation $\mathbf{U}$, where a model is constructed, i.e. a measurement model must be constructed. Once achieved, a parsimonious model for $\mathbf{U}$ as given by (20.1.1) can be reconstructed by sparse regression methods. Importantly, many modern applications of data-driven modelling require that the measurement model and parsimonious model be constructed simultaneously. There are, of course, many nuances to this process, but we will primarily focus at a first approximation on learning spatio-temporal systems governed by PDEs.

Thus, we seek to construct a dynamic model based on data $\mathbf{y} \in \mathbb{R}^m$ measured from a high-dimensional state $\mathbf{U} \in \mathbb{R}^n$, where $n \gg 1$ and $m \ll n$. Here, $\mathbf{y}_k = \mathbf{h}(t_k, \mathbf{x}, \mathbf{u}(t_k)) + \boldsymbol{\sigma}$ and the spatio-temporal dynamics are prescribed by (20.1.1). The frequency of observations are given by $\mathbf{y}_k$, which are measured at the times t_k on some spatial locations prescribed by $\mathbf{x}$. Observations are compromised by measurement noise $\boldsymbol{\sigma}$, which is typically described by a probability distribution (e.g. a normal distribution $\boldsymbol{\sigma} \sim \mathcal{N}(\mu, \sigma)$).

The goal is that given p measurements $\mathbf{y}_k$ arranged in the matrix $\mathbf{Y} = [\mathbf{y}_1\ \mathbf{y}_2\ \cdots\ \mathbf{y}_p] \in \mathbb{R}^{m\times p}$, infer the dynamics $\mathbf{N}(\cdot)$ (or *unknown* portion of dynamics) with parametric dependencies, the measurement operator $\mathbf{h}(\cdot)$, or a proxy model of the true system, so that tasks such as control, characterization, and forecasting can be accomplished. If $\mathbf{h}(\cdot)$ is not the identity and/or $\boldsymbol{\sigma}$ is not zero, then we are in the case of *imperfect data.* This problem can be one of *online* learning where the update must occur in real-time and with no possibility of repeating the experiment. Thus, model discovery is typically an ill-posed problem whose solution must be accomplished through judiciously chosen regularization. Solving this ill-posed problem is a fundamental scientific and mathematical challenge since there are simply not enough constraints on the measurement model, dynamics, and parametrization to achieve a unique solution. Typically, such model discovery has only been accomplished in highly specialized settings, typically with full state measurements and high-quality (low-noise) data. Significant mathematical innovations have to be developed in order to make this a general and robust architecture, especially as regularization is required to make the problem well-posed.

As sufficient data is acquired from sensors in time and space, the data-discovery pipeline then produces the flow

$$\mathbf{y}\ (\text{measurements}) \to \mathbf{u}\ (\text{state space}) \to \mathbf{N}(\cdot)\ (\text{dynamics model}) \tag{20.6.1}$$

with two functions to discover, $\mathbf{h}$ and $\mathbf{N}$. Alternatively, one can also find a new coordinate system $\mathbf{z}$ in which to build the dynamic model so that

$$\mathbf{y}\,(\text{measurements}) \rightarrow \mathbf{u}\,(\text{state space}) \rightarrow \mathbf{z}\,(\text{new state space}) \rightarrow \mathbf{N}(\cdot)\,(\text{dynamic model}) \tag{20.6.2}$$

In addition to the discovery of a measurement model and the underlying dynamics, the dimension of the dynamics r must also be discovered and/or estimated in any data-driven architecture. Often the rank of an underlying subspace on which the physics can be projected can be estimated from the SVD [36]. However, hyper-parameter tuning is critical in refining the initial estimates of the rank r. Such hyper-parameter tuning, which is aimed at justifying the choice of rank through cross-validation, is critical to almost every aspect of training a successful machine learning model. Importantly, we wish to impose physics-based regularization principles in order to make this problem well-posed.

The *sparse identification of nonlinear dynamics* (SINDy) method makes the assumption that the dynamics $\mathbf{N}(\cdot)$ can be represented by only a few terms [159]. In this case, the regression is to the form

$$\frac{d\mathbf{U}}{dt} \approx \boldsymbol{\Theta}\boldsymbol{\xi} \tag{20.6.3}$$

where the columns of the matrix $\boldsymbol{\Theta}$, denoted $\boldsymbol{\theta}_k$, are candidate terms from which to construct a dynamical system or PDE [20, 23, 160]. Example library terms for a scalar, PDE system where $\mathbf{U} \rightarrow U$ are

$$\boldsymbol{\theta}_k = \{U, U^2, U^3, \cdots, U_x, U_{xx}, U_{xxx}, \cdots, U^2U_x, U^2U_{xx}, \cdots\}.$$

The vector $\boldsymbol{\xi}$ contains the loading (or coefficient or weight) of each library term. SINDy assumes that $\boldsymbol{\xi}$ is a sparse vector so that most of the library terms do not contribute to the dynamics, i.e. a parsimonious dynamics is enforced [21, 159, 161, 162]. The regression is a simple $\mathbf{Ax} = \mathbf{b}$ solve for an overdetermined system that is regularized by sparsity, or the sparsity-promoting ℓ_0 or ℓ_1 norms. Indeed, there are a variety of methods for promoting sparsity through optimization. Many of these techniques have been implemented in the PySINDy package [23].

To illustrate the use of the SINDy model discovery method, a PDE example will be considered. For illustrative purposes, we will use the following reaction–diffusion equations as a test model. The coupled field evolution is given by:

$$U_t = 0.1\nabla^2 U + (1 - U^2 - V^2)U + \beta(U^2 + V^2)V \tag{20.6.4a}$$

$$V_t = 0.1\nabla^2 V - \beta(U^2 + V^2)U + (1 - U^2 - V^2)V \tag{20.6.4b}$$

with the inhomogeneous and time-varying parameter

$$\beta = 1 + \sin(2\pi x/L)\sin(2\pi y/L)\sin(2\pi t/10) \tag{20.6.5}$$

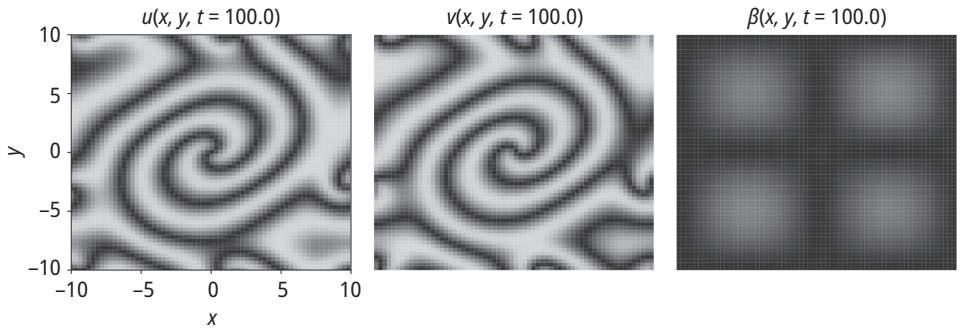


Figure 20.21 Evolution of the reaction–diffusion system with exogenous input field $\beta(x, y, t)$. The data generated from solving this PDE is used in the SINDy architecture to infer the governing evolution equations.

which is included in order to illustrate how to handle exogenous forcing or control signals. Of course, we assume we do not know the actual evolution equations. So the example given is a proof of concept demonstration, or validation, of the SINDy algorithm. Figure 20.21 shows the evolution dynamics of this model.

To generate the data, the governing equations and parametric dependencies are specified as follows:

```
def beta_func(x,y,t, beta0, beta):
    return beta0+beta*np.sin(2*np.pi*x/L)*np.sin(2*np.pi*y/L)
        *np.sin(omega*t)

# Define the reaction-diffusion PDE in the Fourier (kx, ky)
    space
def reaction_diffusion(t, uv, K2, d1, d2, beta0, deltabeta,
    n, N):
    beta = beta_func(x_grid,y_grid,t,beta0,deltabeta)
    u = np.reshape(uv[:N], (n, n))
    v = np.reshape(uv[N:], (n, n))
    u3 = u**3; v3 = v**3
    u2v = (u**2) * v; uv2 = u * (v**2)
    urhs = u - u3 - uv2 + beta * u2v + beta * v3
    vrhs = v - u2v - v3 - beta * u3 - beta * uv2
    u_d = np.real(np.fft.ifft2(-d1 * K2 * np.fft.fft2(u)))
    v_d = np.real(np.fft.ifft2(-d2 * K2 * np.fft.fft2(v)))
    return np.concatenate([(u_d+urhs).reshape(N), (v_d+vrhs).
        reshape(N)])
```

where a spectral solver is then used to advance the solution in time. Spectral PDE solvers are considered previously in the text.

What is assumed is that we do not know the governing equations. Thus, we only know from data collected that

$$U_t = F(U, V, \beta(x, y), x, y, t) \tag{20.6.6a}$$

$$V_t = G(U, V, \beta(x, y), x, y, t) \tag{20.6.6b}$$

where both $F(\cdot)$ and $G(\cdot)$ are unknown, but where we have knowledge of the input $\beta(x, y)$. Our goal is to use the SINDy architecture to discover the right-hand-side terms.

There are many variants of SINDy available today, but it is strongly recommended to use both the weak form of SINDy [163] along with ensembling [164]. These two innovations on SINDy make it a viable tool for real data and the requirements of handling noise. In addition, one selects the highest order of spatial derivatives to consider and the degree of polynomials in the library which we will consider. Here, up to fourth-order polynomials are considered with up to two derivatives. Accurate computations of derivatives higher than second order are very difficult to produce, especially with noisy data. The following code ingests the computational data and regresses to a sparse model:

```
weak_feature_lib = ps.WeakPDELibrary(
    function_library=ps.PolynomialLibrary(degree=4, include_bias=False),
    derivative_order=2,
    spatiotemporal_grid=spatiotemporal_grid,
    H_xt=[dx*5,dx*5,dt*5],
    K=1000,
    include_interaction=False,
)
optimizer = ps.STLSQ(threshold=0.05, alpha=1e-8,
normalize_columns=False)

model = ps.SINDy(feature_library=weak_feature_lib, optimizer=optimizer,
     feature_names=['u','v','beta'])
model.fit(X,u=u,t=t[trimt:])
model.print()
model.score(X,u=u,t=t[trimt:])
```

The optimizer used here is sequential least-square thresholding, which works very well in practice and converges to a Bayesian estimate via ensembling [165, 166]. The discovered SINDy model is as follows:

```
(u)' = 0.989 u - 0.094 u beta + 0.072 v beta - 0.994 u^3 -
     0.977 u v^2
     + 0.107 u^3 beta + 0.915 u^2 v beta + 0.079 u v^2 beta +
         0.917 v^3 beta
     + 0.088 u_22 + 0.090 u_11
(v)' = 0.797 v + 0.061 u v - 0.107 u beta - 0.790 u^2 v -
     0.785 v^3 +
     - 0.055 u^3 v - 0.879 u^3 beta - 0.073 u v^3 - 0.876
         u v^2 beta
     + 0.083 v_22 + 0.086 v_11
```

where $u_1 = u_x, u_{11} = u_{xx}, u_2 = u_y$ and $u_{22} = u_{yy}$. This has the correct terms with close to the right coefficients. However, it does include small amounts of extra terms that could potentially be removed with a little more aggressive thresholding. Currently, the PySINDy package does not forward solve this discovered model automatically. This requires a user to build a PDE solver to test the robustness and stability of the learned model. For dynamical systems, or differential equations, SINDy does offer the capability for evolving the system. However, the differential equation solvers currently used are often suspect and cannot be trusted for evaluating the discovered model.

20.7 Deep Learning Paradigms for Time–Space Stepping

The theme of this chapter largely revolves around the determination of a low-rank subspace in which to build a model for the time dynamics. Thus, we aim to map from measurement space, either full state-space measurements or sparse point measurements, to a latent variable $\mathbf{z}(t)$ in which the time evolution can be represented:

$$\mathbf{x} \text{ or } \mathbf{s} \text{ (measurements)} \rightarrow \mathbf{z} \text{ (latent space)} \rightarrow \tilde{\mathbf{x}} \text{ (reconstruction)}. \tag{20.7.1}$$

Figure 20.22 represents two possibilities for learning efficient representations of the spatio-temporal system. Specifically, an encoder/decoder pairing can be used to map the spatio-temporal data matrix $\mathbf{X}(\mathbf{x}, t)$ to its approximation $\tilde{\mathbf{X}}(\mathbf{x}, t)$. Alternatively, one can sparsely sample from the data matrix $\mathbf{X}(\mathbf{x}, t)$ to make this approximation.

The goal in creating a latent space $\mathbf{z}(t)$ is two-fold. First, the latent space aims to reduce the dimension of the problem to the intrinsic dimension of the dynamics, whether known or unknown. This is a critical step in model reduction generally, but also for learning interpretable coordinates and variables. The second goal is to construct a model for the time evolution of $\mathbf{z}(t)$ itself, which can then be used for *roll-outs* to future states, i.e. forecasting. To begin, consider the autoencoder/decoder structure of the top panel of Fig. 20.22. The deep learning architecture takes the input data $\mathbf{x}_k$ and maps them to the latent space $\mathbf{z}_k$ and then back to an approximation $\tilde{\mathbf{x}}_k$. This gives the compositional structure

$$\tilde{\mathbf{x}}_k = f_{\boldsymbol{\theta}_1}(h_{\boldsymbol{\theta}_2}(f_{\boldsymbol{\theta}_3}(\mathbf{x}_k)))) \tag{20.7.2}$$

where $f_{\boldsymbol{\theta}_3}$ and $f_{\boldsymbol{\theta}_1}$ are the encoder and decoder maps, respectively, and $h_{\boldsymbol{\theta}_2}$ is a latent space model for the time sequence. More generally, the time sequence model is trained so that this mapping gives an approximation to the future state

$$\widetilde{\mathbf{x}}_{k+p} = f_{\boldsymbol{\theta}_1}(h_{\boldsymbol{\theta}_2}(f_{\boldsymbol{\theta}_3}(\mathbf{x}_k))) \tag{20.7.3}$$

where p steps are taken in the time sequence. An obvious part of the loss function is to minimize $\|\tilde{\mathbf{x}}_{k+p} - \mathbf{x}_{k+p}\|$. Recurrent neural networks and their many variants are ideally suited for this task of building a sequential model.

Deep learning time steppers have been developed for model reduction and advancing solutions of spatio-temporal systems. Three papers in particular have advanced time-stepping paradigms whereby the sequential model is more precisely stated as

$$\mathbf{z}_{k+1} = h_{\boldsymbol{\theta}_2}(\mathbf{z}_k, \mathbf{z}_{k-1}, \cdots, \mathbf{z}_{k-p}) \tag{20.7.4}$$

where there are p time lags modelling the dynamics in the RNN structure. Parish and Carlberg [167] and Regazzoni et al. [168] both develop reduced order models using the philosophy of the top panel of Fig. 20.22, where the time-stepping is a sequence model. Parish and Carlberg [167], in fact, provide a detailed comparison of sequence modelling tools for the accuracy of the ROMs. Liu et al. [22] modify the modelling framework by learning a hierarchical time-stepping scheme whereby fast, medium, and slow time scales are explicitly learned separately. This provides a robust and stable algorithm, which is highly effective for sequential modelling. In contrast, neural ODEs learn a neural network map of the dynamics [37–39]. In all these strategies, the neural network learns effectively how to march forward in time in the latent space with a neural network (20.7.4).

Alternatively, one can instead shape the latent space by imposing additional terms in the loss function. In this case, one attempt to discover a linear model by minimizing the loss is

$$\text{argmin}_{\mathbf{A}} \|\mathbf{z}_{k+1} - \mathbf{A}\mathbf{z}_k\|. \tag{20.7.5}$$

Thus, the latent space is forced to learn a linear map between time steps. This strategy was first proposed in Lusch et al. [101] and further developed by Gin et al. [102] in order to construct a linearizing latent space $\mathbf{z}$. This represents a version of Koopman theory learning [169] that is beyond the typical DMD regression. It should be noted that one can instead go up in dimensions in the latent space, but as mentioned already, our goal is to represent interpretable models in the intrinsic rank of the physics itself. Thus, the encoder/decoder structure looks to build the latent space in the smallest possible dimension.

Finally, for the encoder/decoder structure of Fig. 20.22, a SINDy penalty can be prescribed as part of the loss function

$$\text{argmin}_{\boldsymbol{\xi}} \|\dot{\mathbf{z}} - \boldsymbol{\Theta}\boldsymbol{\xi}\| + \lambda\|\boldsymbol{\xi}\|_1 \tag{20.7.6}$$

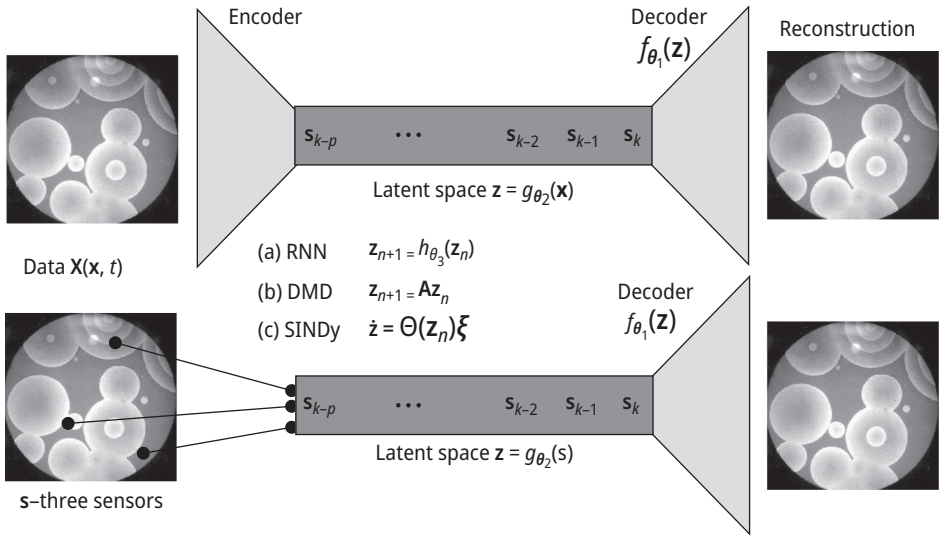


Figure 20.22 Various encoding/decoding and decoding-only architectures for scientific discovery and modelling of spatio-temporal systems. In the top panel, an encoder/decoder pairing is used to construct a latent space $z(t)$ for modelling the time-evolution. In the bottom panel, a decoding only scheme is used from sensor measurements to construct a latent evolution $z(t)$, which maps to the high-dimensional state-space. In both cases, three options are shown for mapping the latent temporal dynamics $z(t)$: (a) map the latent dynamics to a variant of a recurrent neural network, (b) map the latent dynamics to a DMD/Koopman model, or (c) map the latent dynamics to a SINDy model.

where a sparse SINDy fit is attempted on a candidate set of library functions in the latent space. Champion et al. [170] first proposed this architecture in order to learn coordinates and dynamics jointly. Gao et al. [166] extended the results to a Bayesian setting, which is highly effective in handling noisy data and producing uncertainty quantification of the model. Kalia et al. [171] further extended the model learning to enforce normal form dynamics [49] in the latent space in order to capture bifurcation structures.

The same structures can be enforced in the decoding only scheme of the bottom panel of Fig. 20.22. The model given by (20.7.3) is now modified to accept sensor measurements at the input

$$\widetilde{\mathbf{x}}_{k+p} = f_{\boldsymbol{\theta}_1}(h_{\boldsymbol{\theta}_2}(f_{\boldsymbol{\theta}_3}(\mathbf{s}_k))). \tag{20.7.7}$$

The SHRED architecture does away with the encoding step and directly learns a latent space representation from time-delayed measurements. This has proven to be a highly effective strategy. Moreover, the latent space enforcements for a linear model or SINDy can be applied as with the encoder/decoder settings.

The codes for producing the schemes illustrated in Fig. 20.22 can be downloaded from GitHub. Given the various architectural structures and constraints, it is recommended that one directly downloads the code bases to study the implementation strategies.

Data Assimilation Methods

21

21.1 Theory of Data Assimilation 557
21.1.1 Data assimilation for a single random variable 559
21.2 Data Assimilation, Sampling, and Kalman Filtering 562
21.2.1 Relating observations to the state vector 562
21.2.2 The one-dimensional Kalman filter 563
21.2.3 The vector EKF 564
21.3 Data Assimilation for the Lorenz Equation 566
21.3.1 Sensitivity to initial conditions 568
21.3.2 Data assimilation for the Lorenz equations 568

Most of the data-driven techniques presented in this book were applied to systems where the underlying governing equations were prescribed. However, in the DMD method (or in the equation-free method), no governing equations were required to extract meaningful information about the dynamics of the complex system under consideration. The method of *data assimilation* is a hybrid method that uses both data measurements collected in time about the system in conjunction with a prescribed set of governing equations. The fact is, both the measurements and simulations are, in practice, heavily influenced by uncertainty and/or noise fluctuations. Thus, neither the experiment nor the theoretical model can be fully trusted. However, combining the two so that the experimental measurements helps inform the model and vice versa can greatly improve the predictive powers of the model, or analysis of the state of the system [172, 173, 174]. Thus, data is assimilated into the model predictions and hence the name. Data assimilation is potentially one of the most useful data-driven modelling techniques, as we are rarely without some underlying governing equations or without experimental measurements. And to make optimal use of both, data assimilation techniques are ideal.

21.1 Theory of Data Assimilation

To begin thinking about data assimilation, we will first consider a given complex system and its underlying dynamical evolution. Specifically, let us begin by assuming that the system under consideration has the following evolution equation:

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0022

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) \tag{21.1.1}$$

with the initial state of the system given by

$$\mathbf{y}(0) = \mathbf{y}_0 \,. \tag{21.1.2}$$

Of course, techniques for solving such a system have been considered extensively throughout this book. Indeed, provided that $f(t, \mathbf{y})$ is well behaved, i.e. continuous and differentiable, for instance, then a solution to the prescribed problem can be solved for uniquely. In fact, for an N-dimensional vector $\mathbf{y}$, N initial conditions are prescribed by $\mathbf{y}_0$ so that the number of unknowns and constraints match.

Up to this point, we have ignored and/or denied a simple and obvious truth regarding the evolution equation (21.1.1) and its initial conditions (21.1.2). Specifically, we are assuming that we can perfectly prescribe both! In practice, this is surely an impossible task. And for strongly nonlinear systems (21.1.1), even small changes in the initial conditions and/or slight noise perturbations to the system can lead to large changes in the dynamical behaviour, stability and predictability of (21.1.1). To be more precise then about our formulation of the true system, (21.1.1) and (21.1.2) should be modified to

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}) + \mathbf{q}_1(t) \tag{21.1.3}$$

with the initial conditions

$$\mathbf{y}(0) = \mathbf{y}_0 + \mathbf{q}_2 \tag{21.1.4}$$

where $\mathbf{q}_1$ represents unknown model errors due to either noise fluctuations in the system or perhaps truncation of higher-order effects in the system that are thought to be negligible. Similarly, the vector $\mathbf{q}_2$ represents the error in the prescribed initial conditions either because we cannot accurately measure them in practice or prescribe them in a realistic physical system.

Even in the presence of the perturbations $\mathbf{q}_1$ and $\mathbf{q}_2$, the system of equations (21.1.3) and (21.1.4) remains well-posed with a unique solution specifying the dynamics in time. However, for a strongly nonlinear system, the presence of $\mathbf{q}_1$ and $\mathbf{q}_2$ make it virtually impossible to use the governing equations (21.1.3) and the initial conditions (21.1.4) for an accurate prediction of the future state of the system. This is largely due to the concept of *sensitivity to initial conditions* that is displayed in many complex systems. Of course, this then gives rise to the following fundamental question: Is modelling a worthwhile exercise if it cannot predict the future state of a realistic system? In practice, great engineering is often about eliminating large unknowns in a system by essentially suppressing, as much as possible, the vectors $\mathbf{q}_1$ and $\mathbf{q}_2$. But for highly complex systems, such as climate modelling and weather prediction, the system simply cannot be engineered, and one must find effective strategies for dealing with the inherent effect of $\mathbf{q}_1$ and $\mathbf{q}_2$.

Data assimilation is a technique that attempts to mitigate the problems associated with having $\mathbf{q}_1$ and $\mathbf{q}_2$ present in a given system. As the name suggests, the idea is to assimilate

experimental measurements directly into the theoretical model in order to inform the dynamics. Thus, a set of measurements are taken so that

$$g(t, \mathbf{y}) + \mathbf{q}_3 = 0 \tag{21.1.5}$$

where $g(t, \mathbf{y})$ is a certain set of measurements, let us say M of them, on some quantities related to the state vector $\mathbf{y}$, and the vector $\mathbf{q}_3$ is the error measurement associated with the data collection process.

In principle, it is a great idea to incorporate real experimental measurements into the modelling process. In practice, the addition of (21.1.5) now makes for an overdetermined system (N unknowns and $N + M$ constraints) for $\mathbf{y}$ for which no solution exists in general. Thus, the tradeoff of using the experimental data is that we went from a well-posed system with a unique solution to an overdetermined system with no general solution.

To deal with the overdetermined system, the following *quadratic form* is introduced:

$$J(\mathbf{y}) = \int_0^T \int_0^T \mathbf{q}_1^T(t_1)\mathbf{W}_1\mathbf{q}_1(t_2)dt_1dt_2 + \mathbf{q}_2^T\mathbf{W}_2\mathbf{q}_2 + \mathbf{q}_3^T\mathbf{W}_3\mathbf{q}_3 \tag{21.1.6}$$

where the error vectors $\mathbf{q}_j$ are all directly included in the quadratic form. The matrices $\mathbf{W}_j$ are the inverse of the error covariance for the model, initial conditions, and measurements, respectively. The introduction of such a quadratic form is motivated by one primary purpose: optimization. In particular, one potential solution to the overdetermined system is to find the solution that minimizes the weighted error, weighted with respect to the model, initial condition, and measurement error, as given by the quadratic form $J(\mathbf{y})$. Since it is a quadratic form, standard convex optimization methods can be directly applied to find a solution. The least-square error model defined by $J(\mathbf{y})$ is only one potential choice. However, this choice is particularly attractive when considering Gaussian statistics for the error vectors. In this case, minimizing $J(\mathbf{y})$ is equivalent to maximizing the probability density function $P(\mathbf{y}) = C\exp[-J(\mathbf{y})]$. Thus, the minimum of (21.1.6) is also the maximum-likelihood estimate.

Thus, to summarize the data assimilation method, we consider a theoretical model under the influence of some error (21.1.3) subject to initial conditions, which are also subject to error (21.1.4). A number of experimental measurements (21.1.5), also subject to error, are then made to inform the model. The combined errors of the resulting overdetermined system are cast as a quadratic form for which convex optimization techniques can be used to find the best-fit solution (in the L^2 sense). As a result, the experimental measurements are assimilated with the model predictions to generate better predictions of the dynamical behaviour in time. The development of these key ideas will be carried forward in the next two sections.

21.1.1 Data assimilation for a single random variable

To illustrate the ideas of data assimilation, the simplest toy example will be considered (see, for example, the nice arguments by Holton and Hakim [175] in the context of atmospheric

sciences). Specifically, consider a model that generates some prediction about the state of the system, thus effectively $\mathbf{q}_1$ and $\mathbf{q}_2$ would be combined in this model prediction process. Also, consider some experimental measurements on the systems that are also subject to error. Thus, for the variable x, there are two predictions about its true value: one from the model and one from experiment. The idea is to combine these two measurements to arrive at a better prediction of the true value of x.

To begin this calculation, consider the following conditional probability statement from Bayes's formula (see Section 13.2):

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{21.1.7}$$

where $p(x)$ represents the probability density for predicting the variable x. Here, y will represent the observation (experimental assimilation) that will be made. Thus, $p(x|y)$ is the probability of finding x conditioned on having measured y. At this point, we will not concern ourselves with $p(y)$ since it will simply represent a scaling factor for (21.1.7). The probability density function $p(y|x)$ is a *likelihood function* since it is a function of the variable x.

As is the case with almost all problems in probability theory, great simplifications can be made by assuming Gaussian distributed random variables. In fact, our treatment of the data assimilation problem for higher-dimensional problems relies explicitly on this assumption in order to derive something tractable. It also helps simplify the current one-variable problem if this assumption is made. Thus, consider the following probability density distributions:

$$p(y|x) = c_1 \exp\left[-\frac{1}{2}\left(\frac{y-x}{\sigma_y}\right)^2\right] \tag{21.1.8}$$

$$p(x) = c_2 \exp\left[-\frac{1}{2}\left(\frac{x-x_0}{\sigma_0}\right)^2\right] \tag{21.1.9}$$

where σ_y is the error variance for the observation, and x_0, σ_0 are the predicted model mean and its associated error variance. The constants c_1 and c_2 are normalization constants ensuring that probability density functions integrate to unity. Thus the errors, both in measurement (σ_y) and theory (σ_0), are characterized by the variance parameters. Note that without any data assimilation, the model would predict the value x_0. Data assimilation attempts to give a correction to this value using the measurement data.

Using the conditional probability statement, which integrates in information about the observation y, along with the assumed Gaussian probability distributions (21.1.9) yields the following prediction (what is x given observation y?) for the state of the system:

$$p(x|y) = c_3 \exp\left[-\frac{1}{2}\left(\frac{y-x}{\sigma_y}\right)^2\right] \exp\left[-\frac{1}{2}\left(\frac{x-x_0}{\sigma_0}\right)^2\right] \tag{21.1.10}$$

where c_3 is another constant used for convenience.

Our goal now is to construct a quadratic form like (21.1.6) for this problem in order to determine how to modify x from its default prediction value of x_0 in light of our observation y. A very simple quadratic form to construct from the Gaussian distributions is as follows:

$$J(x) = -\log[p(x|y)] + \log(c_3) = \frac{1}{2}\left(\frac{y-x}{\sigma_y}\right)^2 + \frac{1}{2}\left(\frac{x-x_0}{\sigma_0}\right)^2 . \tag{21.1.11}$$

But this quadratic form can be easily minimized, as it is simply a parabola in one dimension, i.e. optimization can be performed in closed form unlike the general formulation (21.1.6). To find its minimum, $dJ(\bar{x})/dx = 0$ is computed. This yields the following value for $\bar{x}$ at the minimum:

$$\bar{x} = \left(\frac{\sigma_y^2}{\sigma_y^2+\sigma_0^2}\right) x_0 + \left(\frac{\sigma_0^2}{\sigma_y^2+\sigma_0^2}\right) y\,. \tag{21.1.12}$$

Thus $\bar{x}$, which is a weighted linear superposition of the model prediction x_0 and the observation y, is the new and improved prediction for the outcome of x given the observation y. Such is the change of value when using the conditional probability argument.

Equation (21.1.12) has some very intuitive features. First and foremost: if there is no error in the experimental observation, then $\sigma_y = 0$ and $\bar{x} = y$. This is essentially a statement of self-consistency. It would be very bad if the data assimilation method failed to choose the observational data value if it was error free. The error variance for $\bar{x}$ can also be computed to be

$$\bar{\sigma}^2 = \frac{\sigma_0^2}{1+\left(\sigma_0^2/\sigma_y^2\right)} = \frac{\sigma_y^2}{1+\left(\sigma_y^2/\sigma_0^2\right)} < \sigma_0^2, \sigma_y^2\,. \tag{21.1.13}$$

This is also a self-consistency check. Namely, the error in the data assimilation prediction is always better than the error of either the model alone or observation alone. Again, the fact that you are using the combination of model and data should always improve your predictions and error.

Figure 21.1 shows the three probability densities of interest. In particular, what is shown is the probability density given by the model, $p(x)$, the observation, $p(y|x)$ and the data

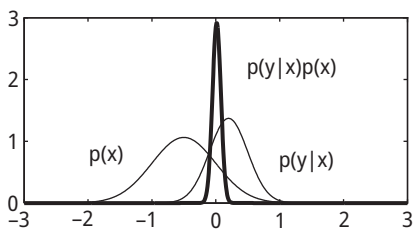


Figure 21.1 Distributions of the model (left Gaussian, $p(x)$), the observation (right Gaussian, $p(y|x)$), and the data assimilated distribution $p(\bar{x}) = p(y|x)p(x)$. Note that the error variance of the assimilated distribution is much narrower than either the model or observational data as shown in (21.1.13).

assimilation, $p(\bar{x}) = p(y|x)p(x)$. For this figure, the following were assumed $x_0 = -0.5$, $\sigma_0 = 0.5$, $y = 0.2$, and $\sigma_y = 0.3$. This allows us to compute the distribution of the assimilated prediction through (21.1.12) and (21.1.13). Note that the variance of the assimilated distribution is quite narrow and shifted strongly towards the observational distribution. This is largely because the observation has a much smaller variance than the model data.

Another way to express these results is by noting that

$$\bar{x} = x_0 + K(y - x_0) \tag{21.1.14}$$

with $\bar{\sigma}^2 = (1 - K)\sigma_0^2$ and where

$$K = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_y^2} \leq 1\,. \tag{21.1.15}$$

The second term in the right-hand side of (21.1.14), $K(y - x_0)$, is the so-called *innovation* since it brings in new information (an observation) to the prediction of x. The predicted value of x is a linear combination of its model prediction x_0 and the innovation. If there is no innovation, i.e. no new information, then the prediction remains x_0. The parameter $0 < K < 1$ is the gain weighting factor and is essentially the so-called *Kalman filter* or *Kalman gain*. This will be considered in more detail in the next section.

21.2 Data Assimilation, Sampling, and Kalman Filtering

The preceding section outlined the high-level view of the data assimilation method and showed how to implement it on the simplest example possible. What is desired now is to develop the theory further for implementation in realistic systems. Of particular note will be the role of the *Kalman filter*, or as already hinted at in the previous section, the method of incorporating *innovation* to the model predictions.

21.2.1 Relating observations to the state vector

Before proceeding to derive the Kalman filter for a general vector system, we address the issue of how observations (21.1.5) project onto the state variable $\mathbf{y}$ of the governing equation (21.1.1). For instance, the governing equation (21.1.1) may be the simulation of some underlying PDE system that is solved on a rectangular grid of a prescribed domain. Observations, however, may be made anywhere in the domain and will, in general, not be aligned with the grid used for (21.1.1). In weather prediction, the data are collected at observation points which may be on the coast, on top of mountains, and/or in metropolitan areas. Certainly, the observation points are not aligned on a regular grid. The simulation of a geographic region, however, is most likely accomplished using a given discretization, perhaps in tens of metres to kilometres. The question is how to overlay the irregularly spaced observations onto the regularly spaced state variables. The simplest thing to do is to use a linear interpolation algorithm to generate values of the state variables at each grid point of the model.

The mapping of the experimental observations can be accomplished mathematically in the following way:

$$\mathbf{y}(t) = \mathbf{H}\mathbf{x}(t) + \mathbf{q}_3 \tag{21.2.1}$$

where $\mathbf{y}(t)$ are the observations at a given time t of the state vector $\mathbf{x}$, $\mathbf{H}$ is a matrix that maps the state vector to the observations, and $\mathbf{q}_3$ is the observational error. This is a linear version of the more general form (21.1.5). This step is always necessary once a data collection has been applied at a time t.

21.2.2 The one-dimensional Kalman filter

The derivation of the projection operator $\mathbf{H}$ in (21.2.1) is necessary for deriving the Kalman filter. To start the derivation process, we once again return to one-dimensional considerations and a highly idealized version of the dynamics in discrete form. In particular, following Miller [172], we can consider the mapping from a time t_k to a time t_{k+1}. The full dynamics, without approximation, is assumed to be given by

$$x_{k+1} = f(x_k) + q_{k+1} \tag{21.2.2}$$

where x_{k+1} and x_k are the state of the one-dimensional system at time t_{k+1} and t_k, respectively, and q_{k+1} is a Gaussian white noise sequence. The model approximation to this system is given by

$$x_{0_{k+1}} = f\left(x_{0_k}\right) \tag{21.2.3}$$

where x_{0_k} is the best estimate of the state of the system at time t_k, sometimes called the *analysis*, and $x_{0_{k+1}}$ is the forecast of the dynamics using this estimate at time t_{k+1}.

The error between the truth and the forecast at time t_{k+1} is then given by

$$x_{k+1} - x_{0_{k+1}} = f(x_k) - f\left(x_{0_k}\right) + q_{k+1}. \tag{21.2.4}$$

By Taylor expanding $f(x_k)$ around x_{0_k}, the above expression can be written as follows:

$$\begin{aligned} x_{k+1} - x_{0_{k+1}} = (x_k - x_{0_k}) f'(x_{0_k}) + \frac{1}{2}(x_k - x_{0_k})^2 f''(x_{0_k}) \\ + \frac{1}{6}(x_k - x_{0_k})^3 f'''(x_{0_k}) + \cdots + q_{k+1}. \end{aligned} \tag{21.2.5}$$

To find the error variance between the correct solution x_{k+1} and our prediction $x_{0_{k+1}}$, the expectation value of the quantity $(x_{k+1} - x_{0_{k+1}})^2$ is computed. Denoting this as $E[(x_{k+1} - x_{0_{k+1}})^2]$, we find, by squaring the above expression and taking the expectation,

$$E\left[(x_{k+1} - x_{0_{k+1}})^2\right] = E\left[(x_k - x_{0_k})^2\right] \left(f'(x_{0_k})\right)^2 + \text{h.o.t} + E\left[q_{k+1}^2\right] \tag{21.2.6}$$

where $\mathrm{h.o.t}$ denotes all higher-order moment terms, i.e. the skewness and kurtosis, for instance, of the error. Keeping these terms represents a closure problem for the error. However, as a first approximation, it is often the case that the higher-order moments are neglected. Thus, we can discard them from the calculation at this point. However, it should be noted that there is a great deal of work that investigates the effect of preserving the higher-order moments in the calculation. One might imagine that the more information is retained, the better the approximation should work.

To simplify the notation, we define the following two quantities:

$$P_{k+1} = E\left[(x_{k+1} - x_{0_{k+1}})^2\right] \tag{21.2.7a}$$

$$P_k = E\left[(x_k - x_{0_k})^2\right] \tag{21.2.7b}$$

which are measures of the error variance between the true solution and the prediction at t_{k+1}, and the true solution and our best estimate of it at t_k, respectively. This notation gives

$$P_{k+1} = P_k(\, f'(x_{0_k}))^2 + E\left[q_{k+1}^2\right]. \tag{21.2.8}$$

Note that P_{k+1} accounts for the dynamics (and errors associated with it), while P_k accounts for errors in estimating the initial state.

To make a data assimilated prediction, $\bar{x}_{k+1}$, of the state of the system at time t_{k+1} using an observation y_{k+1}, we then apply the following formula:

$$\bar{x}_{k+1} = x_{0_{k+1}} + K_{k+1}(y_{k+1} - x_{0_{k+1}}) \tag{21.2.9}$$

where the innovation is again given by the quantity $y_{k+1} - x_{0_{k+1}}$ and the Kalman gain K_{k+1} is given by

$$K_{k+1} = \frac{P_{k+1}}{P_{k+1} + R} \tag{21.2.10}$$

where R is the observation error variance. If there is no observational error, then $R = 0$, which forces $K_{k+1} = 1$. This in turn gives the assimilated prediction to be $\bar{x}_{k+1} = y_{k+1}$, i.e. the assimilated prediction is exactly the error-free experimental observation. The variance associated with the error of our prediction $\bar{x}_{k+1}$ is given by

$$\bar{P}_{k+1} = (1 - K_{k+1})P_{k+1}. \tag{21.2.11}$$

Taken together, the data assimilation results, which now directly tie in the dynamics of the system through (21.2.2), define what is called the *extended Kalman filter* (EKF).

21.2.3 The vector EKF

The vector version of the above one-dimensional argument follows in a straightforward manner [172, 176]. The dynamics given by (21.2.2) and (21.2.3) are now written as

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k) + \mathbf{q}_{k+1} \tag{21.2.12}$$

and

$$\mathbf{x}_{0_{k+1}} = f(\mathbf{x}_{0_k}), \tag{21.2.13}$$

respectively. Once again Taylor expanding now yields the covariance evolution

$$\mathbf{P}_{k+1} = \mathbf{J}(\mathbf{f})\mathbf{P}_k\mathbf{J}(\mathbf{f})^T + \mathbf{Q} \tag{21.2.14}$$

where $\mathbf{J}(\mathbf{f})$ is the Jacobian generated from the multi-dimensional Taylor expansion, and higher-order moments have been neglected. This formula is equivalent to (21.2.8).

Given a data observation $\mathbf{y}_{k+1}$ at time t_{k+1}, the vector case requires an appropriate mapping of the observation space to the state space. But this was already discussed and is mathematically transcribed in (21.2.1). Finally, the data assimilated prediction of the correct state is given by

$$\bar{\mathbf{x}}_{k+1} = \mathbf{x}_{0_{k+1}} + \mathbf{K}_{k+1}\left(\mathbf{y}_{k+1} - \mathbf{H}\mathbf{x}_{0_{k+1}}\right) \tag{21.2.15}$$

where $\mathbf{K}_{k+1}$ is the Kalman gain matrix given by

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1}\mathbf{H}^T(\mathbf{H}\mathbf{P}_{k+1}\mathbf{H}^T + \mathbf{R})^{-1} \tag{21.2.16}$$

and $\mathbf{R}$ is the noise covariance matrix. The error covariance of the updated state vector is given by

$$\bar{\mathbf{P}}_{k+1} = (\mathbf{I} - \mathbf{K}_{k+1}\mathbf{H})\mathbf{P}_{k+1} \tag{21.2.17}$$

where $\mathbf{I}$ is the identity matrix. In vector form, the innovation is given by $\mathbf{y}_{k+1} - \mathbf{H}\mathbf{x}_{0_{k+1}}$. Thus, the matrix $\mathbf{H}$ serves an important role in overlaying the data measurement locations with the underlying grid used for computationally evolving forward the model dynamics. This is the EKF for generic complex systems.

With the EKF now established, its strengths and weaknesses are briefly considered. Its strengths are obvious: the EKF provides a systematic way to integrate observational data into the modelling process, thus improving the model predictions. Its most obvious drawbacks are computational. Specifically, for complex systems where the state vector is defined by potentially millions or billions of variables on a large computational grid, computing the innovation becomes unwieldy, especially as these large matrices need to be inverted and their Jacobians are to be found. Such enormous computational expense can render the method useless from a practical point of view. One potential way to deal with such computational complexity is to minimize (21.1.6) directly using, for instance, gradient descent algorithms. This is ultimately faster than computing Jacobians and inverses of the large matrices under consideration. Another potential way to solve the problem is to use ensemble Kalman filtering (EnKF) techniques, which render the problem tractable by processing observations one at a time or by breaking the domain into smaller subdomains where the matrices are tractable. Such computational considerations are necessary to consider in the types of systems (weather prediction and climate modelling [175]) where data assimilation has become a standard method of analysis.

21.3 Data Assimilation for the Lorenz Equation

To demonstrate the data assimilation algorithm in practice, consideration will be given to the Lorenz equations

$$x' = \sigma(y - x) \tag{21.3.1a}$$

$$y' = rx - y - xz \tag{21.3.1b}$$

$$z' = xy - bz. \tag{21.3.1c}$$

Thus, the vector $\mathbf{x} = [x\ y\ z]^T$ is the three-degree of freedom state vector whose nonlinear evolution $f(\mathbf{x})$ is specified by the right-hand side (Lorenz model) in (21.3.1c). The Lorenz equations are a highly simplified model of convective-driven atmospheric motion.

To begin, a perfect simulation will be performed of the Lorenz system, i.e. a simulation that includes specified initial conditions without noise and an evolution, which is free of stochastic forcing. This simulation will be the *truth* that the data assimilation will try to reproduce. The parameters to be simulated here, and in what follows, are standard in many examples: $\sigma = 10$, $b = 8/3$, and $r = 28$. The large value of r puts the system in a dynamical state that is highly sensitive to initial conditions. The perfect initial conditions will be $\mathbf{x}_0 = [5\ 5\ 5]^T$. The following MATLAB code solves this problem for the time domain $t \in [0, 20]$ and plots it in a parametric way in 3D.

```
t=0:0.01:20;
sigma=10; b=8/3; r=28;

x0=[5 5 5];
[t,xsol]=ode45('lor_rhs',t,x0,[],sigma,b,r)

x_true=xsol(:,1); y_true=xsol(:,2); z_true=xsol(:,3);
figure(1), plot3(x_true,y_true,z_true)
```

The right-hand side of the differential equation includes the dynamics through $f(\mathbf{x})$. In this case, the function **lor_rhs** is given by

```
function rhs=lor_rhs(t,x,dummy,sigma,b,r)
rhs=[sigma*(-x(1)+x(2))
    -x(1)*x(3)+r*x(1)-x(2)
    x(1)*x(2)-b*x(3)];
```

The results of simulating the Lorenz equations are shown in Figs. 21.2 and 21.3. The first of these figures demonstrates the standard *butterfly* pattern of evolution of the strange

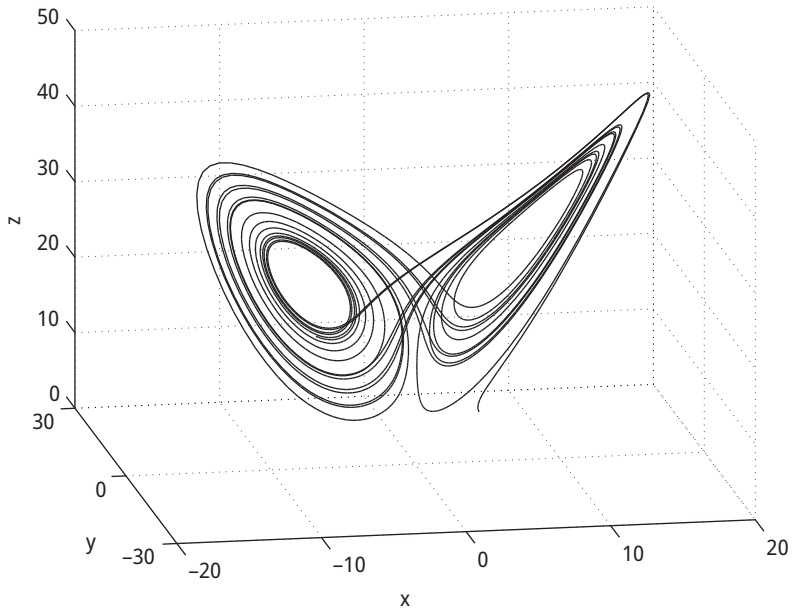


Figure 21.2 Evolution of the variables $x(t)$, $y(t)$, and $z(t)$ over the time period $t \in [0, 20]$ for $\sigma = 10$, $b = 8/3$, and $r = 28$. The evolution is shown parametrically as a function of time.

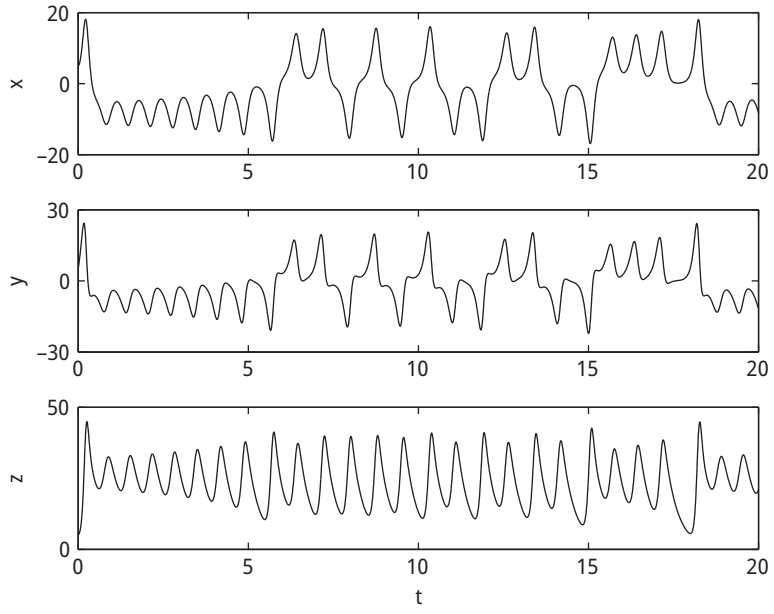


Figure 21.3 Time evolution of the variables $x(t)$, $y(t)$, and $z(t)$ over the time period $t \in [0, 20]$ for $\sigma = 10$, $b = 8/3$, $r = 28$, and $\mathbf{x}_0 = [5\ 5\ 5]^T$. In this parameter regime, specifically for this large a value of r, the dynamics of the Lorenz equations are highly sensitive to initial conditions.

attractor in three dimensions. In this graph, time is a parametric quantity. The second of these figures illustrates the evolution of the variables $x(t)$, $y(t)$, and $z(t)$ over the time period $t \in [0, 20]$. In what follows, instead of tracking and comparing all the variables, we will focus on $x(t)$ for illustrative purposes only.

21.3.1 Sensitivity to initial conditions

The first thing that will be investigated is sensitivity of the evolution to small changes in the initial conditions. The mathematical statement of this problem is given in (21.1.2). Thus, the effect of $\mathbf{q}_2$ on the dynamics will be considered. And in particular, it is the perturbation of the initial conditions that compromises the predictive power of the theoretical model. To simplify this, we will assume that the error has a Gaussian distribution so that

$$\mathbf{x}(0) = \mathbf{x}_0 + \sigma_2 \mathbf{q}(0, 1) \tag{21.3.2}$$

where $\mathbf{x}_0$ is the perfect initial conditions and $\mathbf{q}(0, 1)$ is a Gaussian distributed random variable with mean zero and unit variance. Thus, σ_2 is chosen to make the error variance either larger or smaller.

With this error, the evolution of (21.3.1c) can once again be explored. In Fig. 21.4, eight realizations of the evolution are given using the initial conditions (21.3.2). The exact evolution, which we are trying to model (with $\sigma_2 = 0$) is the thinner solid line, while the initial conditions with error (with $\sigma_2 = 1$) is the bold line. The following MATLAB code generates the eight realizations:

```
sigma2=1;  % error variance
for j=1:8
  xic=x0+sigma2*randn(1,3); % perturb initial conditions
  [t,xsol]=ode45('lor_rhs',t,xic,[],sigma,b,r);
  x=xsol(:,1);  % projected x values
  subplot(4,2,j), plot(t,x_true,'k'), hold on
  plot(t,x,'k','Linewidth',[2])
end
```

For all these realizations, the projected state fails to model the *true* dynamics after $t \approx 5$. Indeed, after this time, there is almost no correlation of closeness between the truth and our projection based upon noisy initial data. This illustrates the need for data assimilation. Specifically, it is hoped that occasional measurements of the data would allow for an accurate prediction of the true future state far beyond $t \approx 5$.

21.3.2 Data assimilation for the Lorenz equations

The goal in this example will be to simply illustrate the EKF algorithm. Thus, a simple case will be taken, which can be completely coded in MATLAB in a fairly straightforward manner.

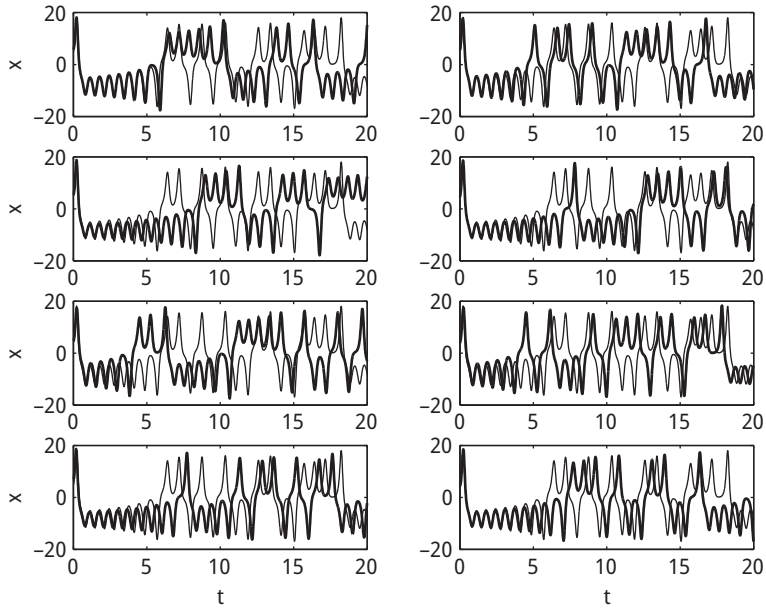


Figure 21.4 Time evolution of the variables $x(t)$, $y(t)$, and $z(t)$ over the time period $t \in [0, 20]$ for $\sigma = 10$, $b = 8/3$, $r = 28$, and $\mathbf{x}_0 = [5\ 5\ 5]^T$. Here, eight realizations are shown for the perturbed initial conditions given by (21.3.2) with $\sigma_2 = 1$. Note that for all simulations, after $t \approx 5$, the true dynamics (light line) differ from the dynamics with perturbed initial conditions (bold line). Thus, prediction of the dynamics beyond this time is virtually impossible given such perturbations (errors) in the initial data.

In this simple example, no stochastic forcing of the differential equations will be considered so that the dynamics are *exactly* as specified in (21.3.1c). Said another way, the error vector $\mathbf{q}_1$ in (21.1.6) is zero. However, there will be error both in the measurements ($\mathbf{q}_3$) and the initial conditions ($\mathbf{q}_2$). Knowing full well about the sensitivity to initial conditions in the Lorenz equations, the initial noise will greatly compromise the ability of the model to predict the future state of the system. The hope is that data assimilation will mitigate this problem to some extent and allow for much more accurate, and longer time, predictions for the future state of the system.

In our example, the data collection points will be at simulation time points already specified by our differential equation solver. Moreover, data will be taken from all variables. Thus, the mapping matrix is $\mathbf{H} = \mathbf{I}$. To illustrate the data collection process, consider then the modification of (21.2.1) to

$$\mathbf{y}(t_n) = \mathbf{x}(t_n) + \sigma_3 \mathbf{q}(0, 1) \tag{21.3.3}$$

where t_n is a measurement time point and $\mathbf{q}(0, 1)$ is a Gaussian distributed random variable with mean zero and unit variance. Thus, σ_3 is chosen to make the error variance either larger or smaller in the data measurements. Figure 21.5 shows the true dynamics (line) and data collected every half-unit of time with $\sigma_3 = 4$ (circles). As is clearly seen, the data are collected

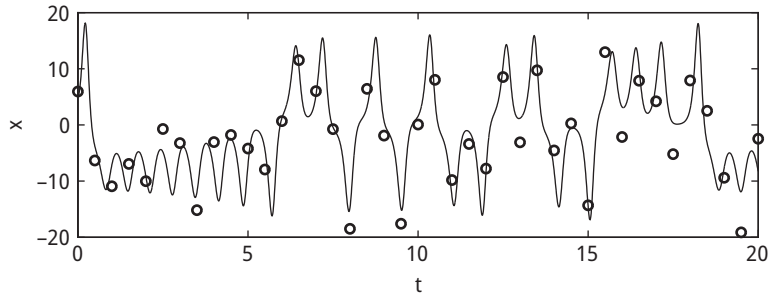


Figure 21.5 True time dynamics of the variable $x(t)$ over the time period $t \in [0, 20]$ for $\sigma = 10$, $b = 8/3$, $r = 28$, and $\mathbf{x}_0 = [5\ 5\ 5]^T$. The circles represent experimental measurements at every half-unit of time of the true dynamics with error given by (21.3.3) with $\sigma_3 = 4$. The dynamics of the variables $y(t)$ and $z(t)$ are similar. Data assimilation makes use of the experimental measurements to keep the model predictions closer to the true dynamics.

under error and do not match up perfectly with the true dynamics. However, they do follow the true dynamics fairly closely over the time period of integration. To compute these data points in MATLAB, the following code is used in conjunction with the code that generates the true dynamics:

```
% noisy obserations every t=0.5
tdata=t(1:50:end);
n=length(tdata)
xn=randn(n,1); yn=randn(n,1); zn=randn(n,1);
sigma3=4;  % error variance in data
xdata=x(1:50:end)+sigma3*xn;
ydata=y(1:50:end)+sigma3*yn;
zdata=z(1:50:end)+sigma3*zn;
```

The idea is to use these experimental points along with the noisy initial conditions shown in Fig. 21.4 in order to enhance our prediction for the future state.

Given that $\mathbf{H} = \mathbf{I}$ in this example, this reduces the EKF algorithm to computing

$$\bar{\mathbf{x}}_{k+1} = \mathbf{x}_{0_{k+1}} + \mathbf{K}_{k+1}\left(\mathbf{y}_{k+1} - \mathbf{x}_{0_{k+1}}\right) \tag{21.3.4}$$

where $\mathbf{K}_{k+1}$ is the Kalman gain matrix given by

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1}(\mathbf{P}_{k+1} + \mathbf{R})^{-1} \tag{21.3.5}$$

and $\mathbf{R}$ is the noise covariance matrix. The error covariance of the updated state vector is given by

$$\bar{\mathbf{P}}_{k+1} = (\mathbf{I} - \mathbf{K}_{k+1})\mathbf{P}_{k+1}\,. \tag{21.3.6}$$

Note that the matrices involved are 3×3 matrices and the innovation is simply given by $\mathbf{y}_{k+1} - \mathbf{x}_{0_{k+1}}$. From (21.2.14), and using the fact that the dynamics propagates in an error-free fashion ($\mathbf{q}_1 = 0$), then

$$\mathbf{P}_{k+1} = \mathbf{J}(\mathbf{f}\,)\mathbf{P}_k\mathbf{J}(\mathbf{f}\,)^T \tag{21.3.7}$$

where the Jacobian for the Lorenz equation can be easily computed to give

$$\mathbf{J}(\mathbf{f}\,) = \begin{bmatrix} -\sigma & \sigma & 0 \\ r-z & -1 & -x \\ y & x & -b \end{bmatrix} \tag{21.3.8}$$

and the matrix $\mathbf{P}_k$ measures the error in estimating the initial state of the system at time t_k. This error is determined by (21.3.2) and the parameter σ_2.

Everything is in place then to implement the data assimilation procedure. The following is an algorithmic outline of what needs to occur:

(1) Determine the sources of error and how to incorporate them. The error in the data measurement determines the matrix $\mathbf{R}$, while the error in the initial condition determines the matrix $\mathbf{P}_k$ (note that we are ignoring errors generated in the dynamics themselves).

(2) Compute the Jacobian at time t_k using the best estimate for the state vector $\mathbf{x}_0(t_k)$, and combine it with the computation of $\mathbf{P}_k$ in order to compute $\mathbf{P}_{k+1}$.

(3) With $\mathbf{P}_{k+1}$ and $\mathbf{R}$, compute the Kalman gain matrix $\mathbf{K}_{k+1}$.

(4) Compute the new state of the system using the innovation vector and the Kalman gain matrix.

(5) Use the new state of the system to again project to another time into the future where observational data is once again available.

The Lorenz equation is a fairly trivial example to consider. Moreover, our treatment of the data assimilation will be for the simplest case possible. In this example, the error variance for both the initial conditions and data measurements will both be unity so that $\sigma_2 = \sigma_3 = 1$, respectively. The Kalman gain matrix will then be given as in the one-dimensional case: $K = \sigma_2/(\sigma_2 + \sigma_3)$. A code for simulating this system and making adjustments based upon the data measurements is as follows:

```
x_da=[];  % data assimilation solution
for j=1:length(tdata)-1  %  step through every t=0.5
  tspan=0:0.01:0.5;  % time between data collection
  [tspan,xsol]=ode45('lor_rhs',tspan,xic,[],sigma,b,r);

  xic0=[xsol(end,1); xsol(end,2); xsol(end,3)]  % model estimate
  xdat=[xdata(j+1); ydata(j+1); zdata(j+1)] % data estimate
  K=sigma2/(sigma2+sigma3);  % Kalman gain
  xic=xic0+(K*[xdat-xic0])  %  adjusted state vector

  x_da=[x_da; xsol(1:end-1,:)];  % store the data
end
x_da=[x_da; xsol(end,:)];  % store last data time
```

In this simulation, the vector **data-xic0** is the innovation vector that is weighted according to the Kalman gain matrix. Figure 21.6 shows the results of this simulation for one representative realization of the error vectors. In the top panels, the nondata-assimilated computation is shown showing that the simulation solution diverges from the true solution around $t \approx 5$ (see also Fig. 21.4). The error between the true solution and the model solution is shown in the right panel. Note that the error is quite large around $t \approx 5$, thus making any prediction

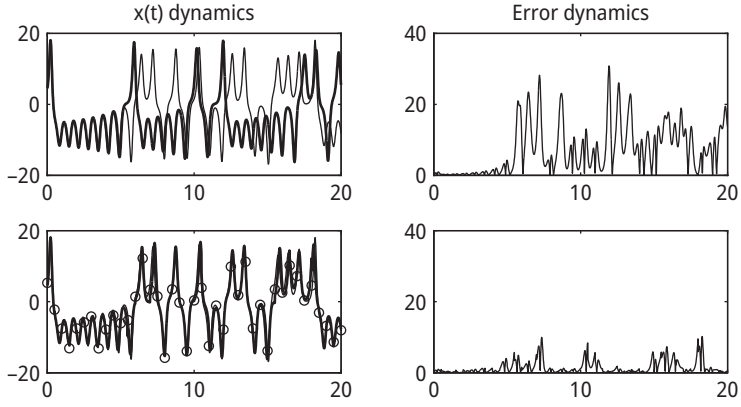


Figure 21.6 Comparison of the model dynamics using a direct numerical simulation of the noisy initial conditions (top panels) with the data assimilated solution with noisy initial conditions and noisy data measurements (bottom panels). The direct simulation (bold line) fails to predict the true dynamics (line) beyond $t \approx 5$ (top panel). Indeed, the error in this case grows quite large at this time (top right panel). When making use of the data assimilation technique (bottom panels) and the data measurements (circles), the solution stays close to the true solution for a much longer time with much smaller error (bottom right panel). Thus, data assimilation can greatly extend the time window under which the model can be useful.

beyond this time fairly useless. In the bottom panel, the data assimilated solution is demonstrated (bold line) and compared to the true dynamics (line). The data assimilated solution is nearly indistinguishable from the true dynamics. The experimental observations are shown by circles at every half-unit of time. The error between the data assimilated solution and true dynamics is shown in the right panel. Note that the error remains quite small in comparison to the direct simulation from noisy initial data. Regardless, there is a build up of error that will eventually grow large as the data assimilated solution also diverges from the true dynamics. Ultimately, the data assimilated solution allows for significant extension of the time window where the model prediction is valid.

Data assimilation, in general, is much more sophisticated than what has been applied here to a simple 3×3 system. Indeed, for highly complex systems, the model error in the dynamics plays a fundamental role characterizing the behaviour in addition to measurement and initial condition errors. How one chooses to not only treat this error, but how to sample from the dynamics in time, gives rise to a great variety of mathematical techniques for enhancing the data assimilation method [172, 173, 174]. Such data assimilation methods are at the heart of cutting-edge technology, for instance, in weather prediction and/or climate modelling. The hope here was simply to illustrate the basic ideas of this tremendously powerful methodology.

22 Transformers and Foundation Models

22.1 Vector Spaces for Word Embeddings 575
22.1.1 Language Models 577
22.1.2 Recurrent Neural Networks 578
22.2 The Transformer and Attention Mechanism 580
22.2.1 Attention Mechanism 582
22.3 Foundation Models for Science and Engineering 585

As already mentioned in the prolegomenon, large language models (LLMs) are becoming ubiquitous across almost every discipline in science and engineering. Empowered by the underlying attention mechanism of the transformer network [1], LLMs are powering ChatGPT (OpenAI), Claude (Anthropic), Gemini (Google), Llama (Meta), etc., as the workhorse architecture for *artificial general intelligence* (AGI). The structure of the transformer, with modifications aimed at specific tasks and application areas, is now being deployed broadly in science and engineering domain applications [2]. The overall vision is that with enough training data in a specific domain, a foundation model can extract knowledge from that field in order to effectively perform a variety of downstream tasks. From time-series prediction [3] to weather forecasting [4], foundation models in different application areas are emerging almost weekly. In this chapter, the goal is to give a high-level overview of how transformer networks are constructed and what the self-attention method enables. The architecture itself is relatively complex in practice, with sophisticated encoding and decoding schemes that empower the overall success. Only superficial coverage will be given here to the structure, but hopefully enough in order to highlight the components necessary for the underpinnings of modern LLMs. Additionally, this chapter is written in Summer of 2025, just before the final publication of the second edition of the book. And given the significantly changing landscape and rapid developments in the domain of LLMs, it is perhaps not the wisest to write a chapter on the subject, as it will be quickly outdated. Regardless, I have chosen to do so anyway because of the broad push across disciplines to develop LLMs in application-specific foundation models. As with most machine learning and AI models, there will be great success achieved, but also a significant amount of exaggeration about how well such foundation models actually work.

Data-Driven Modeling & Scientific Computation. Second Edition. J. Nathan Kutz, Oxford University Press.

© J. Nathan Kutz (2026).

DOI: 10.1093/oso/9780198929093.003.0023

22.1 Vector Spaces for Word Embeddings

To begin thinking about LLMs, we will first consider how words themselves are represented in machine learning. Specifically, the entire data-driven architecture operates on numbers and data that are embedded in vectors or matrices for learning and quantification. Words, however, have meaning and context. Thus, the first challenge in modelling language is to find a numerical representation of words. Once this is achieved, then one can leverage all the optimization tools for back-propagation in deep learning architectures such as PyTorch or JAX.

Before proceeding further, the focus of this book is on scientific computing and scientific machine learning. This would suggest that language embeddings would not play as significant a role in training foundation models. However, modern domain-specific foundation models not only consider quantitative information, but they can also integrate a broader class of information, including annotations and descriptions. This is supplementary information to the typical quantitative measurement data one might typically use in machine learning. Thus, regardless of application, the overall LLM architecture is important to understand, as it can be a generalization of how one typically treats scientific computing and scientific machine learning data streams.

Vectors and matrices are by default what we need in order to embed language with numerical values. Importantly, we also want to preserve semantic similarities between words. Consider the two sentences:

```
Machine learning is great.
Machine learning is awesome.
```

In this example, the words *great* and *awesome* are essentially synonyms. Thus, the idea of semantic similarity is quite common and in traditional language, a thesaurus has been a useful tool to establish similar words (synonyms) and opposite words (antonyms). Thus, when embedding these words, the word *great* and *awesome* should be in similar locations in a vector space. But words can have multiple meanings! Consider the sarcastic and negative sentiment of the following statement using the word *great*:

```
Great, I just failed my exam.
```

In this context, the word *great* should not be associated with the previous use of the word *great*, or the previous use of the word *awesome*. Note that the word *awesome* could have similarly been used in a negative fashion. The goal of a word vector embedding is to give each word a multitude of scores, which are concatenated into a vector space allowing for the myriad of ways that a word might be understood in context. This has led to distributed representations [182], commonly referred to as word embeddings, which map words to dense

vectors in a continuous vector space where semantically similar words are positioned closer together. Popular methods for generating such embeddings include Word2Vec [183], GloVe [184], and more recently, contextual embeddings from transformer-based models like BERT [185].

Word embedding spaces must be learnt through neural network training algorithms whose optimization strategies update neural network weights in order to learn contextual numerical representations of words. These learnt word embedding representations are typically obtained through unsupervised training on a large corpora of text. Specifically, modern LLMs learn on the entire internet of data, or corpus of human knowledge. The training process encourages the embedding vectors to capture relationships between words based on their surrounding context. For instance, in Word2Vec's skip-gram model, the objective is to predict the context words given a target word, while in the continuous bag-of-words (CBOW) model, the goal is to predict a target word from its surrounding context. The resulting embeddings not only allow for efficient input into neural networks but also encode rich semantic information that can be exploited for a variety of downstream tasks. In scientific applications, these embeddings can be fine-tuned or augmented to represent specialized terminology, ensuring that domain-specific nuances are preserved and utilized effectively in LLM architectures.

Importantly, although these embedding structures have been developed for encoding words, the concept of creating an embedding space is quite a bit more general. Indeed, the ideas of PCA, DMD, and encode/decoder architectures already are tasked with such representational conceptions. In what we have considered previously in this book, the latent space models we constructed had some measure of interpretability that could be exploited. Here, it is not obvious what interpretability means in the context of language, as they do not evolve, as far as we know, according to governing equations. However, the behaviour of the word embedding space does have features, which we might say are interpretable. Specifically, similar contextual words are closely embedded in a vector space, much as a thesaurus would give for synonyms. Similarly, the opposites of words (antonyms) might be embedded far away in the word vector space. The fact that deep neural networks can learn such embeddings is remarkable. It is also the reason foundation models are on the rise in every single scientific and engineering field: the LLM architecture is capable of learning an exceptionally general latent representation that encodes useful information in proper embedding representation that is often beyond our simple interpretable understanding. As will be shown, transformers are simply a sophisticated encoder–decoder pairing.

As an example of what an embedding looks like in three components of the embedding space, consider Fig. 22.1 which illustrates what an embedding space might be like in three of the latent variable dimensions. The number of latent variables used is prescribed by the size of hidden layers in the neural network. The network originally starts with a random vector for each representation of a word, but after training, each word has a new vector $\mathbf{w}_j$ for the jth word. There are many tricks used for learning such embeddings, including efficient ways to represent words and train models. Importantly, algorithms like Word2Vec also learn words that are around the word of interest; thus, the n words before and after a particular word is used. This is called bag of words. The embedding then explicitly learns words that

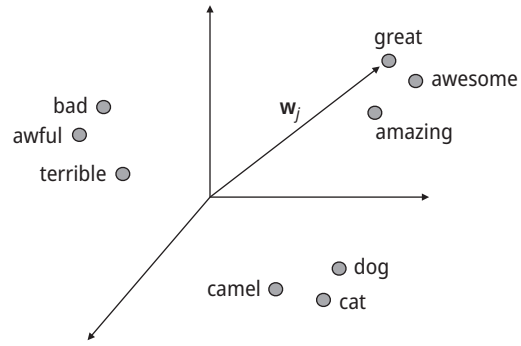


Figure 22.1 Illustration of a word embedding space projected into three dimensions of the latent representation. Note that the learnt embedding space groups similar words in context with each other. This transformation from words with semantic meaning to a representation in vector space is a critical first step in language and speech processing.

are commonly used in conjunction with a specific word. This becomes especially important in learning word prediction.

22.1.1 Language Models

Word embeddings are the *coordinate* system for building something much more interesting: language models. Language models are about making sense of language more broadly in terms of how sequences of words are constructed to create meaning. The simple way to think about a language model is to consider a sequence of words where the jth word of the sequence is given by

$$\mathbf{w}_j \; - \; j\text{th word} \tag{22.1.1}$$

and $\mathbf{w}$ is the vector representation of the word in an embedding space. The simple goal of language is to predict the probability of the next word in a sequence, given the previous n words. Thus,

$$p(\mathbf{w}_j|\mathbf{w}_{j-1}, \mathbf{w}_{j-2}, \cdots \mathbf{w}_{j-n}) \tag{22.1.2}$$

is the probability of the jth word conditioned on the n previous words. As can be seen, the word embedding representation $\mathbf{w}_j$ provides the numerical vector quantification necessary for computing such probabilities. From our previous representation of neural networks, we can state this as learning weights of the neural network

$$\mathbf{w}_j = f_{\boldsymbol{\theta}}(\mathbf{w}_{j-1}, \mathbf{w}_{j-2}, \cdots \mathbf{w}_{j-n}) \tag{22.1.3}$$

where $f_\theta(\cdot)$ is a specific neural network whose weights are given by $\boldsymbol{\theta}$. As is standard, the weights of the neural network are updated in training via back-propagation and some form of gradient descent.

As an example, consider being given the start phrase:

```
Hello, how are
```

which we can consider being represented in the embedding space as $\mathbf{w}_{j-3}$ (hello), $\mathbf{w}_{j-2}$ (how), and $\mathbf{w}_{j-1}$ (are). With a trained sequence model, the goal is to then predict the next word

$$\mathbf{w}_j = f_{\boldsymbol{\theta}}(\mathbf{w}_{j-1}, \mathbf{w}_{j-2}, \cdots \mathbf{w}_{j-n}) \;\rightarrow\; f_{\boldsymbol{\theta}}(\mathbf{w}_{j-1}, \mathbf{w}_{j-2}, \mathbf{w}_{j-3}) \tag{22.1.4}$$

which would give with high probability

$$\mathbf{w}_j \;\rightarrow\; \text{you.} \tag{22.1.5}$$

With the most probable next word computed, one can feed this back into the neural network to produce the next word, and then the next word, etc. Thus, you might unroll the neural network into the future to produce the following sequence with high probability:

```
Hello, how are - you?  I am fine.  How about you?
```

One can imagine that eventually there would be problems continuing this particular line of thought. Transformers, however, allow for much more flexible unrolling architectures if enough key words are given. Indeed, the modern transformer architectures have the capability of looking over the entire corpus of the training data (which is the entire internet) in order to construct a response or narrative from an initial prompt.

22.1.2 Recurrent Neural Networks

One of the methods used to model word sequences in the early days of language modelling was the *recurrent neural network* (RNN) and its variants that include the long-short term memory (LSTM) [128] and gated recurrent neural networks (GRU) [129]. These neural networks were structured to generically solve the sequence-to-sequence model

$$\mathbf{x}_j = f_{\boldsymbol{\theta}}(\mathbf{x}_{j-1}, \mathbf{x}_{j-2}, \cdots \mathbf{x}_{j-n}) \tag{22.1.6}$$

where $\mathbf{x}_j$ is the state-space of given model ($\mathbf{x}_j = \mathbf{w}_j$ for word embeddings), n is the look back (lag or time-history), $f_\theta(\cdot)$ is a specific neural network (RNN, LSTM, GRU) whose weights are given by $\boldsymbol{\theta}$. The weights of the neural network are updated in training via back-propagation and some form of gradient descent. RNNs were originally quite successful in building language models. By training the model (22.1.6), the statistically most likely state-space representation was given from a history of n previous times, or words. Although successful, all the variants of RNN, LSTM, GRU were limited in their success due to the limited memory, or look back, parameter n. Thus, the allowable window for producing the next statistical word

was of length n. In addition, the order of the sequence was quite important in the neural network model $f_{\boldsymbol{\theta}}(\mathbf{x}_{j-1}, \mathbf{x}_{j-2}, \cdots, \mathbf{x}_{j-n})$.

Thus, the following structure of the RNN was explicitly constructed by design: (i) The RNNs were specifically designed to work with data where the order of elements is crucial, such as text, audio, or time series data, (ii) the RNNs learnt a latent, or hidden-state representation of the sequence. This allowed the RNN to maintain context and relationships between elements in the sequence and also acts as a memory for past inputs, and (iii) the recurrent connections in the RNN enabled the hidden state to be updated at each time (sequence) step, incorporating information from previous inputs. Unlike traditional neural networks where weights are different for each node, RNNs share the same weights across different time steps, reducing the number of parameters and making them efficient for sequential data. RNNs, LSTMs, and GRUs are now standard in PyTorch and JAX architectures; thus, they can be executed with almost no barrier to entry. It should be noted that the hidden-state representation simply represents another form of encoding. Thus, the word encoding can first take language to a vector representation $\mathbf{w}_j$, and the RNN further builds an encoding in terms of its latent (hidden state) representation. Ultimately, the success of deep learning is primarily about good encoding and decoding schemes in almost any domain of application.

To be more precise about the mathematical architecture of an RNN, the neural network architecture is designed to model sequential data by incorporating temporal dynamics through recurrence. Unlike feed-forward networks, RNNs maintain a hidden state that is updated at each time step, allowing the network to capture dependencies across time. Formally, for a sequence of inputs $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$, the RNN computes a sequence of hidden states $\{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_N\}$ using the recurrence relation:

$$\mathbf{h}_t = \boldsymbol{\phi}(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h),$$

where $\boldsymbol{\phi}$ is a nonlinear activation function, such as the $\tanh$ or ReLU, $\mathbf{W}_{xh}$ is the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (recurrent) weight matrix, and $\mathbf{b}_h$ is the bias vector.

The output at each time step can be computed using the hidden state:

$$\mathbf{y}_t = \boldsymbol{\psi}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y),$$

where $\mathbf{W}_{hy}$ is the hidden-to-output weight matrix, $\mathbf{b}_y$ is the output bias, and $\boldsymbol{\psi}$ is typically a softmax (for classification) or identity function (for regression). All the weight matrices and biases are learnt during training on a corpus of data. This can be language, but it can also simply be time-series data. The key advantage of RNNs is their ability to share parameters across time, enabling them to generalize across sequence lengths. However, basic RNNs are prone to vanishing and exploding gradients when learning long-term dependencies due to repeated matrix multiplications during back-propagation through time. More advanced RNN variants such as LSTMs and GRUs introduce gating mechanisms that control information flow. These gates—input, forget, and output—allow the network to retain or discard information selectively, enabling the modelling of long-range dependencies. Despite these advancements, RNNs are gradually being supplanted in many applications by attention-based architectures like transformers, which can model sequences without recurrence and

with greater computational efficiency. Nonetheless, understanding RNNs remains essential for understanding the evolution and mathematical underpinnings of sequential deep learning models.

22.2 The Transformer and Attention Mechanism

The attention mechanism included in the transformer network is not new. In fact, attention paradigms in neural networks were inspired by neuronal networks in biology. Thus, beginning in the 1980s, various computer vision applications began implementing different forms of attention, with one of the earliest being *selective attention* [186] by Fukushima who also formulated the original neocognitron [187]. Since this early work, there has been quite an impressive range of achievements associated with the attention mechanisms and its variants [188]. A full history is beyond the scope of this book, but it is simply noted that attention has been an important structure in neural networks for approximately four decades.

What is new is the use of the transformer architecture, which has the attention mechanism directly embedded in the learning structure [1]. The paper, entitled "*attention is all you need*," outlines the transformer architecture, which allows for sequence modelling without recurrence or convolution. By doing away with the RNN architecture for predicting next words $\mathbf{w}_j$, the transformer also eliminates the significant constraints of the memory length n that is a hyper-parameter in RNNs, LSTMs, and GRUs, for instance. In fact, one can think of a transformer as having an *infinite* memory as training of the transformer with the attention mechanism essentially allows the entire corpus of human knowledge to help make the next word prediction. Specifically, during the training process, each word is considered in context of other words that are associated with it throughout the entire training data set, not just in the last n sequence points of the memory. Thus, when making a next word prediction, the attention allows for computing the probability of the next word from the entire corpus of training data.

As noted previously, the transformer architecture is an encoder–decoder pairing, which can be abstractly represented in Fig. 22.2. By constructing these two fundamental blocks,

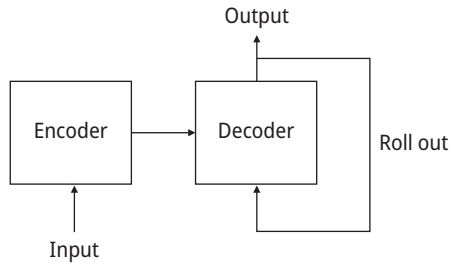


Figure 22.2 Basic structure of a language model. A new sentence or language sequence is input into the encoder, which then predicts the next word $\mathbf{w}_j$ or words in the sequence. The next word can then be decoded back to language structure. The next word is also used to *unroll* the neural network and predict entire new sentences and paragraphs by repeated feeding back into the neural network of the new words predicted, i.e. $\mathbf{w}_j$ gives $\mathbf{w}_{j+1}$, which gives $\mathbf{w}_{j+2}$ and so forth.

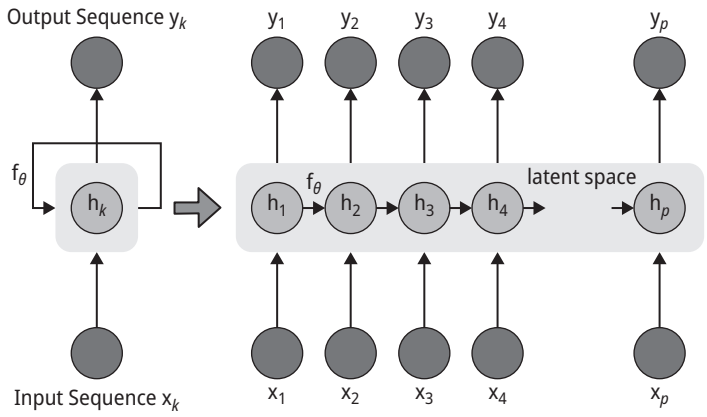


Figure 22.3 Recurrent neural network structure which trains on sequences of input data $\mathbf{x}_k = \mathbf{x}(t_k)$ and output data $\mathbf{y}_k = \mathbf{y}(t_k)$. Unlike a feed-forward neural network, the time history of the sequence, or memory, is used to train the neural network. Thus, the output of the neural network is fed back into the latent layer $\mathbf{f}_{\boldsymbol{\theta}}$. The left representation of the RNN shows the recurrent structure that is achieved from feeding the output back into the neural network. The right representation is the *unfolding* of the graph, which shows a neural network $\mathbf{f}_{\boldsymbol{\theta}}$ that shares weights across different time points.

one can input an initial language sequence into the encoder to decode to the next word in its latent representation. The next word is then used to *unroll*, in a probabilistic way, the neural network forward in the sequence. For example, here is an initial prompt into ChatGPT: **As aliens entered our planet**. The resulting *unroll* of this produces the following:

> *As aliens entered our planet, the skies turned a luminous violet, and a strange silence fell across the world. Cities stood still—air traffic grounded, highways emptied—as millions gazed upward in awe and fear. The ships hovered silently, vast and geometric, casting long shadows over oceans and forests alike. Unlike anything imagined in science fiction, these beings did not announce themselves with weapons or words. Instead, they began releasing glowing orbs into the atmosphere—some kind of autonomous probes that moved with purpose, gently landing across ecosystems, cities, and scientific institutions. Every government scrambled to decode the orbs' intention, but they emitted no threat, only patterns of light and harmonics too complex to be human-made.*

This is an example of an unroll where each word $\mathbf{w}_j$ produces the probability for the next word $\mathbf{w}_{j+1}$, which unrolls as long as desired. In this case, I requested one paragraph from ChatGPT. Typically, limits are placed upon the unrolling from a tokenization limitation, which is how words are encoded in the LLMs (approximately speaking, words are tokens). If we run the exact same prompt again, then we get a different unrolling:

> *As aliens entered our planet, the sky split with a soundless shimmer, like light bending through glass. They came not in warships but in elegant structures that floated effortlessly, their surfaces pulsing with shifting patterns. The world watched in stunned silence as these vessels positioned*

themselves over oceans, deserts, and mountains—never touching the ground, never making demands. Governments scrambled jets, scientists pointed every sensor skyward, and news stations ran endless loops of shaky footage. But the aliens made no hostile moves. Instead, they released a signal—subtle but undeniable—received by every radio, satellite, and even through the static of old televisions. It was music: haunting, mathematical, and strangely beautiful.

This illustrates the probabilistic nature of the transformer architecture. In both these cases, the aliens did not have intentions on our destruction, which is odd given the number of books probably used in training, which would have them come to enslave or destroy us. In fact, after a dozen prompts, I could not have it deliver a simple story where they arrived and blew us up, which suggests ChatGPT was trained to be nice. Indeed, one of the remarkable features of modern LLMs is how well they have been trained to be respectful and not potty-mouthed in their interactions despite being trained on the internet.

22.2.1 Attention Mechanism

The attention mechanism, for which there is a diversity of different structures [189], allows one to capture contextual meaning of a given word and its dependency on other elements of an input sequence. Specifically, any given word is given context from the words around it. Moreover, unlike the RNN, LSTM, GRU sequence structures, the attention mechanism can learn the context over the entire data set by learning three key matrices: queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$. These matrices are learnt over the corpus of the training data, which allows for *infinite* histories versus the finite history of the standard RNN models. In essence, when predicting the next word $\mathbf{w}_j$, it only relies on the current word $\mathbf{w}_{j-1}$ and any context cues preceding it. For instance, adjectives, adverbs, and verbs all modify the meaning of a noun in a sentence and give it context. And of course, all the weights for the matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are updated through training on a large body of text, which allows for the computation of probabilities for the next word given the current word and those modifying it. Importantly, the transformer architecture has a critical *positional embedding* step in order to understand the order of the embeddings. This in large part allows us to move away from the limited history to an infinite memory.

Before proceeding to a highlight of the mathematical methods, a more detailed description of the transformer architecture is given. Specifically, a detailed view of the encoder/decoder structure is detailed in Fig. 22.4. This highlights both the encoder itself as well as the decoder which is ultimately responsible for the unrolling of the LLM. Figure 22.4 (left panel) shows how the encoder is constructed. An input stream is given to the encoder for processing. To make this more mathematically concrete, consider the input as a sequence of embedded vectors

$$\mathbf{X} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_p]^T \in \mathbb{R}^{p\times d} \tag{22.2.1}$$

where p is the sequence length and d is the model embedding dimension, which is typically large for tokenized language embeddings (e.g. for the GPT3 model a value of $d = 12228$ is used). Each input position is first embedded and then augmented with a positional encoding

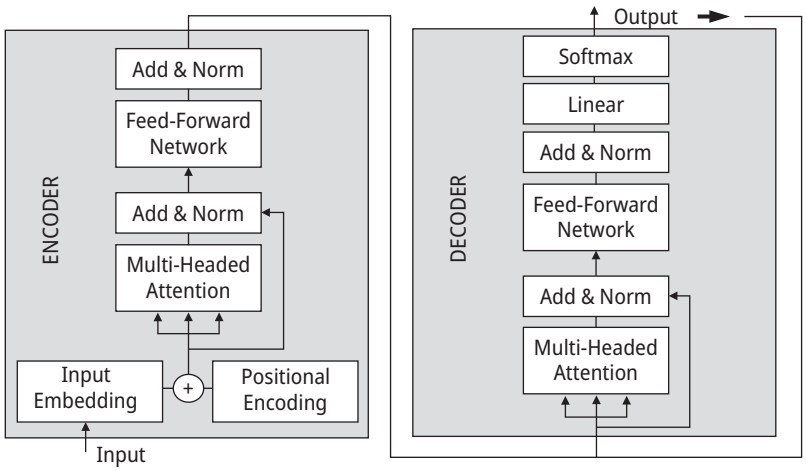


Figure 22.4 The transformer encoder and decoder schemes. The architecture first embeds the inputs into a word embedding space and provides a positional encoding for the words. The encoding then applies multi-headed attention with appropriate normalizations (see Fig. 22.5) to learn context and meaning for the input stream. The decoder module also has a multi-headed attention unit for learning context and meaning in word structures. The output of the decoder is the next word $\mathbf{w}_j$. This can then be recursively fed back into the decoder to produce a *roll out*.

to inject order information. The positional encoding is critical, as it helps understand how a sequence of words should be put together. For example, adjectives are generally before nouns, i.e. **red dog** versus **dog red**. The joint embedding and positional encoding are then fed into a multi-headed attention module before some normalization and feed-forward neural network steps. The attention head module is detailed in Fig. 22.5. The attention head is composed of three learnt matrices:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q \qquad \text{queries} \tag{22.2.2a}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^K \qquad \text{keys} \tag{22.2.2b}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^V \qquad \text{values} \tag{22.2.2c}$$

where $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d\times d_k}$, and $\mathbf{W}^V \in \mathbb{R}^{d\times d}$, where $d_k \ll d$ (e.g. $d_k = 128$). The query is the word itself, while the keys are typically modifiers to the key. For example, the keys matrix gives the probability of words being associated with the key. For instance, the words like **big, red, fluffy** are all high values in the keys matrix for the word **dog**. Whereas words like **skiing, headphones, baking** would have low key matrix scores. The words most associated with **dog** are said to *attend to* that word. Again, all of these weights are learnt during the training process. Moreover, there are multiple attention heads so that an ensemble of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are learnt during training. Since each matrix is initialized randomly, this gives a number of matrices, which can all be used to produce the probability of the next word in the sequence. The scaled-dot product attention multiplies the queries against keys $\mathbf{QK}$, which essentially

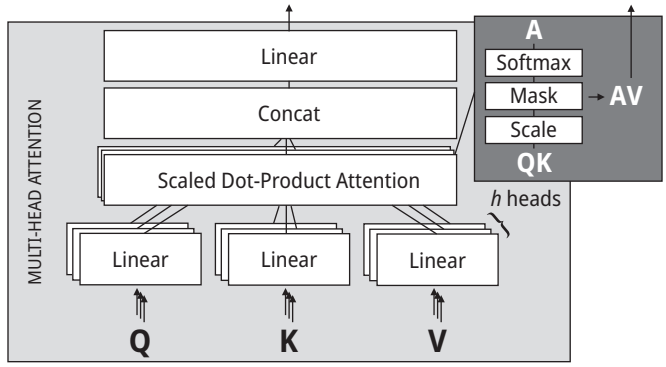


Figure 22.5 Details of the multi-head attention architecture. The attention mechanism learns context and meaning by construction of the three key matrices: queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$. The normalization ensures the matrices behave like probability distributions and masking ensures that a *casual* relationship is preserved in which future words do not influence the choice of the next word. Thus, only past words influence the next word.

computes the probability of each query against each key value. Masking and a soft mask can then produce a new matrix $\mathbf{A}$ by scaled dot multiplication with the value matrix $\mathbf{V}$. The output of this is fed through normalization layers and a feed-forward network, which is then input to the decoder block. The value matrix $\mathbf{V}$ provides a further refinement of the contextual meaning of a word by understanding how surrounding words shift a given word (query) in the embedding space. Thus, given a word like **animal**, a modifier like **slimy animal** would shift the original embedding of the word to a new part of the embedding space where things like slugs and snails might exist. Alternatively, a modifier like **fluffy animal** would shift the embedding space of the word to a place where dogs, cats, gerbils, etc. might be clustered in the embedding space. Thus, the value function helps further learn the refinement of a word and its context by giving specific shifts of the embedding that make sense and that are learnt during training on a large corpus of textual information.

The decoder block mimics the structure of the encoder. However, the output of the encoder is already in the latent space used by the decoder. Thus the data, which already has a positional encoding is decoded by going through a multi-headed attention layer where again an ensemble of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are learnt during training. These are once again normalized, via softmax and scaling, and then fed through a last feed-forward neural network for producing a probability distribution for the next word. The next word can then be fed back into the decoder in order to *unroll* the neural network and produce new text strings. One only needs to specify how long the unroll is, and the neural network will give stop upon the request. In principle, one can ask the LLM to produce an infinite text and it will simply keep *unrolling* in the way shown Fig. 22.4.

The self-attention mechanism and its mathematical operations are represented in the following mathematical form:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \tag{22.2.3}$$

where the dot-product between queries and keys captures alignment scores, and the scaling by $\sqrt{d_k}$ stabilizes gradients. The softmax normalizes these scores into attention weights and produces what is essentially a probability distribution where the sum of distributions is unity.

To enhance the model's ability to focus on information from different representation subspaces, the transformer uses *multi-head attention*, thus an ensemble of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are learnt during training. This involves computing attention outputs across h different heads, each with its own learnt projection matrices. For the i-th head, we compute:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V),$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ are the parameters for head i. The outputs of all heads are concatenated and projected:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O,$$

where $\mathbf{W}^O \in \mathbb{R}^{hd_k \times d}$ is a learnt output projection matrix. Each Transformer layer consists of a multi-head self-attention sublayer followed by a position-wise feed-forward network. These sublayers are wrapped in residual connections and layer normalization. Formally, the full layer output is:

$$\text{LayerOutput} = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V})),$$

$$\text{FFNOutput} = \text{LayerNorm}(\text{LayerOutput} + \text{FFN}(\text{LayerOutput})),$$

where $\text{FFN}(\mathbf{z}) = \max(\mathbf{0}, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$ is a two-layer feed-forward network applied to each position independently. Stacking multiple such layers allows the transformer to learn hierarchical sequence representations. The transformer model has become foundational in natural language processing, computer vision, and scientific modelling due to its scalability and expressivity.

The power of the multi-head attention is to create an exceptionally rich contextual embedding of words in an embedding space, which has a meaningful representation of language. The exceptional success of LLMs shows that an amazing paradigm for learning of context has been established. While the transformer model is black-box, there are some clear interpretability aspects. For instance, the word embedding space has many attractive features, which show that clustering happens in this space for words that have similar semantic meanings. Additionally value vectors can very nicely manipulate this latent space in order to again achieve interpretable projections in the word embedding space [190, 191, 192]. In more recent efforts, embedding interpretable symbolic regression on the attention heads themselves allows for further interpretable methods for the transformer architecture [193].

22.3 Foundation Models for Science and Engineering

As noted previously, there is tremendous flexibility in embedding any kind of data into an LLM transformer architecture. Once the tokenization has been established, the

data can then be put through multi-headed attention units to produce desired outcomes. For instance, instead of prediction of a next word, one might instead use the architecture for time-series forecasting to predict the value of set of measurements. Time series modelling and forecasting is perhaps the easiest analogue in science and engineering with next word prediction. As a result, in a very short amount of time there have been numerous foundation models proposed for time-series prediction [3, 194, 195, 196, 197, 198, 199, 200, 201] and/or spatio-temporal dynamics [202, 203]. These foundation models represent the application of transformer LLM architectures to scientific and engineering data. What has arisen is not only a large number of groups developing innovations and perturbations of the basic transformer structure, but also a large number of domain foundation models based upon groups having large amounts of specific domain data.

While it has become clear that transformers have revolutionized language, it is not so clear that foundation models are anywhere near the performance levels to revolutionize a given application domain, especially as the primary goal in science and engineering is often extrapolation. It is certainly not clear that the data required to train the likes of ChatGPT (OpenAI), Claude (Anthropic), Gemini (Google), Llama (Meta), etc. is even available in most domain applications. Indeed, very few of the above foundation models even bother to compare their performance to each other, let alone well-developed leading traditional methods. This will produce a credibility crisis in science and engineering until a common task framework structure is developed to fairly assess the methods being developed. It is important to remember that foundation models are *fundamentally interpolation engines*! Thus, expecting a foundation model or LLM to predict unobserved physics phenomenon (extrapolation) is in general a flawed premise.

Perhaps in the third edition of this text, a deeper dive into transformers and foundation models will be given. The field is currently moving too fast to write in detail about any specific method. Within the next few years, more clarity will come as the methods will be properly be tested and benchmarked. But at least for now, the reader should have a sketch of how the transformer and attention mechanisms are formulated, and potentially how they can be modified for scientific purposes.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[3] Gerald Woo, Chenghao Liu, Akshat Kumar, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Unified training of universal time series forecasting transformers. *arXiv preprint arXiv:2402.02592*, 2024.

[4] Cristian Bodnar, Wessel P Bruinsma, Ana Lucic, Megan Stanley, Johannes Brandstetter, Patrick Garvan, Maik Riechert, Jonathan Weyn, Haiyu Dong, Anna Vaughan, et al. Aurora: A foundation model of the atmosphere. *arXiv preprint arXiv:2405.13063*, 2024.

[5] Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1):38, 2019.

[6] Nick McGreivy and Ammar Hakim. Weak baselines and reporting biases lead to overoptimism in machine learning for fluid-related partial differential equations. *arXiv preprint arXiv:2407.07218*, 2024.

[7] RL Burden, and JD Faires. *Numerical Analysis*. Brooks/Cole, 1997.

[8] Anne Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.

[9] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain. 1994.

[10] L Sirovich, and M Kirby. A low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America A*, 4(3):519–524, 1987.

[11] M Kirby, and L Sirovich. Application of the Karhunen-Loève procedure for the characterization of human faces. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 12(1):103–108, 1990.

[12] M Turk, and A Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[13] Grace Wahba. Smoothing noisy data with spline functions. *Numerische Mathematik*, 24(5):383–393, 1975.

[14] Peter Craven, and Grace Wahba. Smoothing noisy data with spline functions. *Numerische Mathematik*, 31(4):377–403, 1978.

[15] Abraham Savitzky, and Marcel JE Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36(8):1627–1639, 1964.

[16] Emad Fatemi Leonid I Rudin, and Stanley Osher. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60:259–268, 1992.

[17] Rick Chartrand. Numerical differentiation of noisy, nonsmooth data. *ISRN Applied Mathematics*, 164564, 2011.

[18] Greg Welch, Gary Bishop, et al. An introduction to the Kalman filter, 1995.

[19] Bethany Lusch, Eric C Chi, and J Nathan Kutz. Shape constrained tensor decompositions. In *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 287–297. IEEE, 2019.

[20] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[21] Joshua L Proctor, Steven L Brunton, Bingni W Brunton, and JN Kutz. Exploiting sparsity and equation-free architectures in complex systems. *The European Physical Journal Special Topics*, 223(13):2665–2684, 2014.

[22] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 267–288, 1996.

[23] Alan Kaptanoglu, Brian de Silva, Urban Fasel, Kadierdan Kaheman, Andy Goldschmidt, Jared Callaham, Charles Delahunt, Zachary Nicolaou, Kathleen Champion, J Kutz, et al. PySINDy: A comprehensive python package for robust sparse system identification. *Journal of Open Source Software*, 7(69):3994, 2022.

[24] Brian M de Silva, Kathleen Champion, Markus Quade, Jean-Christophe Loiseau, J Nathan Kutz, and Steven L Brunton. PySINDy: a Python

package for the sparse identification of nonlinear dynamics from data. *Journal of Open Source Software*, 5(49):2104, 2020.

[25] Jeffrey C Lagarias, James A Reeds, Margaret H Wright, and Paul E Wright. Convergence properties of the Nelder–Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998.

[26] David G Luenberger, Yinyu Ye, et al. *Linear and nonlinear programming*, volume 2. Springer, 1984.

[27] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[28] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford University Press, 1995.

[29] William E Boyce, Richard C DiPrima, and Douglas B Meade. *Elementary differential equations and boundary value problems*. John Wiley & Sons, 2021.

[30] Richard Courant, Fritz John, Albert A Blank, and Alan Solomon. *Introduction to calculus and analysis*, volume 1. Springer, 1965.

[31] Steven M Cox and Paul C Matthews. Exponential time differencing for stiff systems. *Journal of Computational Physics*, 176(2):430–455, 2002.

[32] AK Kassam, and LN Trefethen. Fourth-order time-stepping for stiff PDEs. *SIAM Journal on Scientific Computing*, 26(4):1214–1233, 2005.

[33] KW Mahmud, JN Kutz, and WP Reinhardt. Bose–Einstein condensates in a one-dimensional double square well: Analytical solutions of the nonlinear schrödinger equation. *Physical Review A*, 66(6):063607, 2002.

[34] Yuying Liu, J Nathan Kutz, and Steven L Brunton. Hierarchical deep learning of multiscale differential equation time-steppers. *Philosophical Transactions of the Royal Society A*, 380(2229):20210200, 2022.

[35] Tong Qin, Kailiang Wu, and Dongbin Xiu. Data driven governing equations approximation using deep neural networks. *Journal of Computational Physics*, 395:620–635, 2019.

[36] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

[37] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[38] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[39] Patrick Kidger. On neural differential equations. *arXiv preprint arXiv:2202.02435*, 2022.

[40] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge University Press, 2007.

[41] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[42] John C Strikwerda. *Finite difference schemes and partial differential equations*. SIAM, 2004.

[43] Edwin Ding, and J Nathan Kutz. Operating regimes, split-step modeling, and the Haus Master mode-locking model. *JOSA B*, 26(12):2290–2300, 2009.

[44] Gilbert Strang. *Introduction to applied mathematics*. SIAM, 1986.

[45] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

[46] Bernard Deconinck, and J Nathan Kutz. Computing spectra of linear operators using the Floquet–Fourier–Hill method. *Journal of Computational Physics*, 219(1):296–321, 2006.

[47] George William Hill. On the part of the motion of the lunar perigee which is a function of the mean motions of the sun and moon. 1886.

[48] Ali H Nayfeh and Dean T Mook. *Nonlinear oscillations*. John Wiley & Sons, 2008.

[49] Philip Holmes, and John Guckenheimer. *Nonlinear oscillations, dynamical systems, and bifurcations of vector fields*, volume 42. *Applied Mathematical Sciences*. Springer-Verlag, Berlin, Heidelberg, 1983.

[50] Izrail Moiseevitch Gelfand, Richard A Silverman, et al. *Calculus of variations*. Courier Corporation, 2000.

[51] Lokenath Debnath. *Wavelet transforms and time–frequency signal analysis*. Springer Science & Business Media, 2012.

[52] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, 1910.

[53] Tony F Chan and Jianhong Shen. *Image processing and analysis: variational, PDE, wavelet, and stochastic methods*. SIAM, 2005.

[54] Cleve Moler. "magic" reconstruction: Compressed sensing. *Mathworks News & Notes*, 2010.

[55] EJ Candès, and MB Wakin. An introduction to compressive sampling. *IEEE Signal Processing Magazine*, 21–30, 2008.

[56] RG Baraniuk. Compressive sensing. *IEEE Signal Processing Magazine*, 24(4):118–120, 2007.

[57] DL Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[58] Lloyd N Trefethen, and David Bau III. *Numerical linear algebra*, 50. SIAM, 1997.

[59] P Holmes, JL Lumley, G Berkooz, and CW Rowley. *Turbulence, coherent structures, dynamical systems and symmetry*. Cambridge University Press, Cambridge, 2nd paperback edition, 2012.

[60] JP Cusumano, MT Sharkady, and BW Kimble. Experimental measurements of dimensionality and spatial coherence in the dynamics of a flexible-beam impact oscillator. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 347(1683):421–438, 1994.

[61] BF Feeny, and R Kappagantu. On the physical interpretation of proper orthogonal modes in vibrations. *Journal of sound and vibration*, 211(4):607–616, 1998.

[62] Timothy Michael Kubow. *Principal component analysis of cockroach kinematics supports the hypothesis that running is one dimensional*. PhD thesis, UCSF, 2007.

[63] R Ruotolo, and Cecilia Surace. Damage assessment of multiple cracked beams: Numerical results and experimental validation. *Journal of Sound and Vibration*, 206(4):567–588, 1997.

[64] Kevin L Briggman, Henry DI Abarbanel, and WB Kristan Jr. Optical imaging of neuronal populations during decision-making. *Science*, 307(5711):896–901, 2005.

[65] Joshua B Tenenbaum, Vin de Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[66] ST Roweis and LK Saul. Linear embedding nonlinear dimensionality reduction by locally. *Science*, 290(5500):2323–2326, 2000.

[67] Kilian Q Weinberger, and Lawrence K Saul. Unsupervised learning of image manifolds by semidefinite programming. *International Journal of Computer Vision*, 70:77–90, 2006.

[68] EJ Candès, X Li, Y Ma, and J Wright. Robust principal component analysis? *Journal of the ACM*, 58(3):11–1–11–37, 2011.

[69] Zhouchen Lin, Minming Chen, and Yi Ma. The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices. *arXiv preprint arXiv:1009.5055*, 2010.

[70] Zhouchen Lin, Arvind Ganesh, John Wright, Leqin Wu, Minming Chen, and Yi Ma. Fast convex optimization algorithms for exact recovery of a corrupted low-rank matrix. *Coordinated Science Laboratory Report no. UILU-ENG-09-2214, DC-246*, 2009.

[71] Jian-Feng Cai, Emmanuel J Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010.

[72] Xiaoming Yuan, and Junfeng Yang. Sparse and low-rank matrix decomposition via alternating direction methods. *Preprint*, 12(2), 2009.

[73] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: Data-driven modeling of complex systems*. SIAM, 2016.

[74] PJ Schmid. Dynamic mode decomposition for numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 2010.

[75] PJ Schmid, L Li, MP Juniper, and O Pust. Applications of the dynamic mode decomposition. *Theoretical and Computational Fluid Dynamics*, 25(1–4):249–259, 2011.

[76] CW Rowley, I Mezić, S Bagheri, P Schlatter, and DS Henningson. Spectral analysis of nonlinear flows. *Journal of Fluid Mechanics*, 645:115–127, 2009.

[77] Kevin K Chen, Jonathan H Tu, and Clarence W Rowley. Variants of dynamic mode decomposition: Boundary condition, Koopman, and Fourier analyses. *Journal of Nonlinear Science*, 22(6):887–915, 2012.

[78] Cecile Penland. Random forcing and forecasting using principal oscillation pattern analysis. *Monthly Weather Review*, 117(10):2165–2185, 1989.

[79] Cécile Penland, and Ludmila Matrosova. Expected and actual errors of linear inverse

model forecasts. *Monthly Weather Review*, 129(7):1740–1745, 2001.

[80] Michael A Alexander, Ludmila Matrosova, Cécile Penland, James D Scott, and Ping Chang. Forecasting Pacific SSTs: Linear inverse model predictions of the pdo. *Journal of Climate*, 21(2):385–402, 2008.

[81] Meghana Velegar, N Benjamin Erichson, Christoph A Keller, and J Nathan Kutz. Scalable diagnostics for global atmospheric chemistry using ristretto library (version 1.0). *Geoscientific Model Development*, 12(4):1525–1539, 2019.

[82] Travis Askham and J Nathan Kutz. Variable projection methods for an optimized dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 17(1):380–416, 2018.

[83] Diya Sashidhar and J Nathan Kutz. Bagging, optimized dynamic mode decomposition for robust, stable forecasting with spatial and temporal uncertainty quantification. *Philosophical Transactions of the Royal Society A*, 380(2229):20210199, 2022.

[84] JN Kutz, SL Brunton, BW Brunton, and JL Proctor. *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*. SIAM, 2016.

[85] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC Press, 1984.

[86] Joshua L Proctor, Steven L Brunton, and J Nathan Kutz. Dynamic mode decomposition with control. *SIAM Journal on Applied Dynamical Systems*, 15(1):142–161, 2016.

[87] Steven L Brunton, Joshua L Proctor, Jonathan H Tu, and J Nathan Kutz. Compressed sensing and dynamic mode decomposition. *Journal of Computational Dynamics*, 2(2):165–191, 2016.

[88] N Benjamin Erichson, Steven L Brunton, and J Nathan Kutz. Compressed dynamic mode decomposition for background modeling. *Journal of Real-Time Image Processing*, 16(5):1479–1492, 2019.

[89] Alessandro Alla and J Nathan Kutz. Nonlinear model order reduction via dynamic mode decomposition. *SIAM Journal on Scientific Computing*, 39(5):B778–B796, 2017.

[90] Francesco Andreuzzi, Nicola Demo, and Gianluigi Rozza. A dynamic mode decomposition extension for the forecasting of parametric dynamical systems. *SIAM Journal on Applied Dynamical Systems*, 22(3):2432–2458, 2023.

[91] JN Kutz, X Fu, and SL Brunton. Multi-resolution dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 15(2):713–735, 2016.

[92] Kathleen P Champion, Steven L Brunton, and J Nathan Kutz. Discovery of nonlinear multiscale systems: Sampling strategies and embeddings. *SIAM Journal on Applied Dynamical Systems*, 18(1):312–333, 2019.

[93] Sara M Ichinaga, Francesco Andreuzzi, Nicola Demo, Marco Tezzele, Karl Lapo, Gianluigi Rozza, Steven L Brunton, and J Nathan Kutz. PyDMD: A python package for robust dynamic mode decomposition. *arXiv preprint arXiv:2402.07463*, 2024.

[94] JH Tu, CW Rowley, DM Luchtenburg, SL Brunton, and JN Kutz. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

[95] Daniel Dylewsky, David Barajas-Solano, Tong Ma, Alexandre M Tartakovsky, and J Nathan Kutz. Dynamic mode decomposition for forecasting and analysis of power grid load data. *arXiv preprint arXiv:2010.04248*, 2020.

[96] John Ferré, Ariel Rokem, Elizabeth A Buffalo, J Nathan Kutz, and Adrienne Fairhall. Non-stationary dynamic mode decomposition. *IEEE Access*, 2023.

[97] Steven L Brunton, Marko Budisic, Eurika Kaiser, and J Nathan Kutz. Modern Koopman theory for dynamical systems. *SIAM Review*, 64(2):229–340, 2022.

[98] JD Cole. On a quasi-linear parabolic equation occurring in aerodynamics. *Quarterly of Applied Mathematics*, 9:225–236, 1951.

[99] Eberhard Hopf. The partial differential equation $u_t + uu_x = \mu u_{xx}$. *Communications on Pure and Applied mathematics*, 3(3):201–230, 1950.

[100] Mark J Ablowitz, and Harvey Segur. *Solitons and the inverse scattering transform*, 4. SIAM, 1981.

[101] Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature Communications*, 9(1):4950, 2018.

[102] Craig Gin, Bethany Lusch, Steven L Brunton, and J Nathan Kutz. Deep learning models for global coordinate transformations that linearise PDEs. *European Journal of Applied Mathematics*, 1–25, 2020.

[103] Samuel E Otto, Nicholas Zolman, J Nathan Kutz, and Steven L Brunton. A unified framework to

enforce, discover, and promote symmetry in machine learning. *arXiv preprint arXiv:2311.00212*, 2023.

[104] Ronald R Coifman, and Stéphane Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30, 2006.

[105] SL Brunton, BW Brunton, JL Proctor, E Kaiser, and JN Kutz. Chaos as an intermittently forced linear system. *Nature Communications*, 8(19):1–9, 2017.

[106] Hassan Arbabi, and Igor Mezic. Ergodic theory, dynamic mode decomposition, and computation of spectral properties of the Koopman operator. *SIAM Journal on Applied Dynamical Systems*, 16(4):2096–2126, 2017.

[107] Seth M Hirsh, Sara M Ichinaga, Steven L Brunton, J Nathan Kutz, and Bingni W Brunton. Structured time-delay models for dynamical systems with connections to Frenet–Serret frame. *Proceedings of the Royal Society A*, 477(2254):20210097, 2021.

[108] Matthew J Colbrook. The multiverse of dynamic mode decomposition algorithms. *arXiv preprint arXiv:2312.00137*, 2023.

[109] Matthew O Williams, Ioannis G Kevrekidis, and Clarence W Rowley. A data-driven approximation of the Koopman operator: extending dynamic mode decomposition. *Journal of Nonlinear Science*, 6:1307–1346, 2015.

[110] Matthew O Williams, Clarence W Rowley, and Ioannis G Kevrekidis. A kernel approach to data-driven Koopman spectral analysis. *Journal of Computational Dynamics*, 2(2):247–265, 2015.

[111] F Takens. Detecting strange attractors in turbulence. *Lecture Notes in Mathematics*, 898:366–381, 1981.

[112] Enrico Bozzo, Roberto Carniel, and Dario Fasino. Relationship between singular spectrum analysis and fourier analysis: Theory and application to the monitoring of volcanic activity. *Computers & Mathematics with Applications*, 60(3):812–820, 2010.

[113] J Nathan Kutz, Joshua L Proctor, and Steven L Brunton. Applied Koopman theory for partial differential equations and data-driven modeling of spatio-temporal systems. *Complexity*, 2018(1):6010634, 2018.

[114] N Benjamin Erichson, Sergey Voronin, Steven L Brunton, and J Nathan Kutz. Randomized matrix decompositions using r. *Journal of Statistical Software*, 89:1–48, 2019.

[115] N Halko, PG Martinsson, and JA Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[116] William B Johnson, and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[117] Daniel Ahfock, William J Astle, and Sylvia Richardson. Statistical properties of sketching algorithms. *arXiv preprint arXiv:1706.03665*, 2017.

[118] N Benjamin Erichson, Lionel Mathelin, J Nathan Kutz, and Steven L Brunton. Randomized dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 18(4):1867–1891, 2019.

[119] Travis Askham, Peng Zheng, Aleksandr Aravkin, and J Nathan Kutz. Robust and scalable methods for the dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 21(1):60–79, 2022.

[120] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[121] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[122] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49. JMLR Workshop and Conference Proceedings, 2012.

[123] Richard Everson and Lawrence Sirovich. Karhunen–Loeve procedure for gappy data. *JOSA A*, 12(8):1657–1664, 1995.

[124] Jan P Williams, Olivia Zahn, and J Nathan Kutz. Sensing with shallow recurrent decoder networks. *arXiv preprint arXiv:2301.12011*, 2023.

[125] Megan R Ebers, Jan P Williams, Katherine M Steele, and J Nathan Kutz. Leveraging arbitrary mobile sensor trajectories with shallow recurrent decoder networks for full-state reconstruction. *arXiv preprint arXiv:2307.11793*, 2023.

[126] Shervin Sahba, Christopher C Wilcox, Austin McDaniel, Benjamin D Shaffer, and J Nathan Kutz. Shallow recurrent decoder for compressed aero-optical wavefront sensing. In

Laser Science, JW4A–64. Optica Publishing Group, 2023.

[127] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[128] Sepp Hochreiter, and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[129] Rahul Dey, and Fathi M Salem. Gate-variants of gated recurrent unit (GRU) neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 1597–1600. IEEE, 2017.

[130] Mantas Lukoševičius. A practical guide to applying echo state networks. In *Neural Networks: Tricks of the Trade: Second Edition*, 659–686. Springer, 2012.

[131] Francesco Giuliari, Irtiza Hasan, Marco Cristani, and Fabio Galasso. Transformer networks for trajectory forecasting. In *2020 25th International Conference on Pattern Recognition (ICPR)*, 10335–10342. IEEE, 2021.

[132] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2114–2124, 2021.

[133] N Benjamin Erichson, Lionel Mathelin, Zhewei Yao, Steven L Brunton, Michael W Mahoney, and J Nathan Kutz. Shallow neural networks for fluid flow reconstruction with limited sensors. *Proceedings of the Royal Society A*, 476(2238):20200097, 2020.

[134] Aapo Hyvärinen, and Erkki Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4–5):411–430, 2000.

[135] Hany Farid and Edward H Adelson. Separating reflections from images by use of independent component analysis. *JOSA A*, 16(9):2136–2145, 1999.

[136] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[137] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[138] Aapo Hyvärinen, and Erkki Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4–5):411–430, 2000.

[139] M Chris Jones and Robin Sibson. What is projection pursuit? *Journal of the Royal Statistical Society: Series A (General)*, 150(1):1–18, 1987.

[140] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[141] Yann LeCun. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[142] Stuart Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[143] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal statistical Society. Series B (Methodological)*, 1–38, 1977.

[144] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7(2):179–188, 1936.

[145] Tianqi Chen, and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794, 2016.

[146] Gérard Biau and Erwan Scornet. A random forest guided tour. *Test*, 25:197–227, 2016.

[147] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM, 1992.

[148] Corinna Cortes, and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[149] Thomas M Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, (3):326–334, 1965.

[150] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[151] Matthew O Williams, Eli Shlizerman, Jon Wilkening, and J Nathan Kutz. The low dimensionality of time-periodic standing waves

in water of finite and infinite depth. *SIAM Journal on Applied Dynamical Systems*, 11(3):1033–1061, 2012.

[152] James P Gordon, and Hermann A Haus. Random walk of coherently amplified solitons in optical fiber transmission. *Optics Letters*, 11(10):665–667, 1986.

[153] JN Kutz, and Ping Kong Alexander Wai. Ideal amplifier spacing for reduction of gordon-haus jitter in dispersion-managed soliton communications. *Electronics Letters*, 34(6):522–523, 1998.

[154] Clarence W Rowley and Jerrold E Marsden. Reconstruction equations and the Karhunen–Loève expansion for systems with symmetry. *Physica D: Nonlinear Phenomena*, 142(1):1–19, 2000.

[155] J Nathan Kutz, Maryam Reza, Farbod Faraji, and Aaron Knoll. Shallow recurrent decoder for reduced order modeling of plasma dynamics. *arXiv preprint arXiv:2405.11955*, 2024.

[156] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, 5301–5310. PMLR, 2019.

[157] Jiazhong Mei and J Nathan Kutz. Long sequence decoder network for mobile sensing. *arXiv preprint arXiv:2407.10338*, 2024.

[158] Alfio Quarteroni, Andrea Manzoni, and Federico Negri. *Reduced Basis Methods for Partial Differential Equations: An Introduction*, volume 92. Springer, 2015.

[159] J Nathan Kutz and Steven L Brunton. Parsimony as the ultimate regularizer for physics-informed machine learning. *Nonlinear Dynamics*, 107(3):1801–1817, 2022.

[160] SH Rudy, SL Brunton, JL Proctor, and JN Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(e1602614), 2017.

[161] Seth M Hirsh, David A Barajas-Solano, and J Nathan Kutz. Sparsifying priors for bayesian uncertainty quantification in model discovery. *Royal Society Open Science*, 9(2):211823, 2022.

[162] Jared L Callaham, James V Koch, Bingni W Brunton, J Nathan Kutz, and Steven L Brunton. Learning dominant physical processes with data-driven balance models. *Nature Communications*, 12(1):1016, 2021.

[163] Daniel A Messenger, and David M Bortz. Weak SINDy for partial differential equations. *Journal of Computational Physics*, 443:110525, 2021.

[164] Urban Fasel, J Nathan Kutz, Bingni W Brunton, and Steven L Brunton. Ensemble-SINDy: Robust sparse model discovery in the low-data, high-noise limit, with active learning and control. *Proceedings of the Royal Society A*, 478(2260):20210904, 2022.

[165] L Gao, Urban Fasel, Steven L Brunton, and J Nathan Kutz. Convergence of uncertainty estimates in ensemble and Bayesian sparse model discovery. *arXiv preprint arXiv:2301.12649*, 2023.

[166] L Mars Gao and J Nathan Kutz. Bayesian autoencoders for data-driven discovery of coordinates, governing equations and fundamental constants. *Proceedings of the Royal Society A*, 480(2286):20230506, 2024.

[167] Eric J Parish and Kevin T Carlberg. Time-series machine-learning error models for approximate solutions to parameterized dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 365:112990, 2020.

[168] Francesco Regazzoni, Luca Dede, and Alfio Quarteroni. Machine learning for fast and reliable solution of time-dependent differential equations. *Journal of Computational Physics*, 397:108852, 2019.

[169] Steven L Brunton, Marko Budišić, Eurika Kaiser, and J Nathan Kutz. Modern Koopman theory for dynamical systems. *arXiv preprint arXiv:2102.12086*, 2021.

[170] Kathleen Champion, Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, 2019.

[171] Manu Kalia, Steven L Brunton, Hil GE Meijer, Christoph Brune, and J Nathan Kutz. Learning normal form autoencoders for data-driven discovery of universal, parameter-dependent governing equations. *arXiv preprint arXiv:2106.05102*, 2021.

[172] Robert N Miller, Michael Ghil, and Francois Gauthiez. Advanced data assimilation in strongly nonlinear dynamical systems. *Journal of Atmospheric Sciences*, 51(8):1037–1056, 1994.

[173] Pierre Gauthier. Chaos and quadri-dimensional data assimilation: A study based on the Lorenz model. *Tellus A: Dynamic Meteorology and Oceanography*, 44(1):2–17, 1992.

[174] Geir Evensen. Advanced data assimilation for strongly nonlinear dynamics. *Monthly Weather Review*, 125(6):1342–1354, 1997.

[175] James R Holton and Gregory J Hakim. *An introduction to dynamic meteorology*, volume 88. Academic press, 2013.

[176] Andrew H Jazwinski. *Stochastic processes and filtering theory*. Courier Corporation, 2007.

[177] Gerald B Folland. *Introduction to partial differential equations*. Princeton University Press, 1995.

[178] George Keith Batchelor. *An introduction to fluid dynamics*. Cambridge University Press, 2000.

[179] Peter Benner, Serkan Gugercin, and Karen Willcox. A survey of projection-based model reduction methods for parametric dynamical systems. *SIAM Review*, 57(4):483–531, 2015.

[180] Athanasios C Antoulas. *Approximation of large-scale dynamical systems*. SIAM, 2005.

[181] Jan S Hesthaven, Gianluigi Rozza, Benjamin Stamm, et al. *Certified reduced basis methods for parametrized partial differential equations*. Springer, volume 590, 2016.

[182] Gerald Woo, Chenghao Liu, Akshat Kumar, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Unified training of universal time series forecasting transformers. *arXiv preprint arXiv:2402.02592*, 2024.

[183] Cristian Bodnar, Wessel P Bruinsma, Ana Lucic, Megan Stanley, Johannes Brandstetter, Patrick Garvan, Maik Riechert, Jonathan Weyn, Haiyu Dong, Anna Vaughan, et al. Aurora: A foundation model of the atmosphere. *arXiv preprint arXiv:2405.13063*, 2024.

[184] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[185] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[186] Zellig S Harris. Distributional structure. *Word*, 10(2–3):146–162, 1954. Taylor & Francis.

[187] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[188] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[189] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the ACL: Human Language Technologies*, pages 4171–4186, 2019.

[190] Derya Soydaner. Attention mechanism in neural networks: where it comes and where it goes. *Neural Computing and Applications*, 34(16):13371–13385, 2022. Springer.

[191] Kunihiko Fukushima. Neural network model for selective attention in visual pattern recognition and associative recall. *Applied Optics*, 26(23):4985–4992, 1987. Optical Society of America.

[192] F Fukushima. A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[193] Alexey Yermakov, David Zoro, Mars Liyao Gao, and J Nathan Kutz. T-SHRED: Symbolic regression for regularization and model discovery with transformer shallow recurrent decoders. *Philosophical Transactions of the Royal Society A*, 2026.

[194] Bryan Lim, Sercan Ömer Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021. Elsevier.

[195] Shi Bin Hoo, Samuel Müller, David Salinas, and Frank Hutter. The tabular foundation model TabPFN outperforms specialized time series forecasting models based on simple features. In *NeurIPS Workshop on Time Series in the Age of Large Models*, 2024.

[196] Abdul Fatir Ansari, Lorenzo Stella, Caner Turkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Sundar Rangapuram, Sebastian Pineda Arango, Shubham Kapoor, et al. Chronos: Learning the language of time series. *arXiv preprint arXiv:2403.07815*, 2024.

[197] Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. Are transformers effective for time series forecasting? In *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(9):11121–11128, 2023.

[198] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020. Elsevier.

[199] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew G Wilson. Large language models are zero-shot time series forecasters. *Advances in Neural Information Processing Systems*, 36:19622–19635, 2023.

[200] Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, et al. Time-LLM: Time series forecasting by reprogramming large language models. *arXiv preprint arXiv:2310.01728*, 2023.

[201] Jeffrey Lai, Anthony Bao, and William Gilpin. PANDA: A pretrained forecast model for universal representation of chaotic dynamics. *arXiv preprint arXiv:2505.13755*, 2025.

[202] Maximilian Herde, Bogdan Raonic, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for PDEs. *Advances in Neural Information Processing Systems*, 37:72525–72624, 2024.

[203] Michael McCabe, Bruno Régaldo-Saint Blancard, Liam Parker, Ruben Ohana, Miles Cranmer, Alberto Bietti, Michael Eickenberg, Siavash Golkar, Geraud Krawezik, Francois Lanusse, et al. Multiple physics pretraining for spatiotemporal surrogate models. *Advances in Neural Information Processing Systems*, 37:119301 119335, 2024.

Index

dct, 396
ode45, 242
randintrlv, 396
spdiags, 242
annotation, 154
avifile, 167
axis, 34
bicgstab, 46, 51, 116, 234
colormap, 157
cov, 327
cumtrapz, 83
dblquad, 84
disp, 28
double, 58, 382
dwt2, 498
eigs, 54
eig, 54
fft2, 228
fftshift, 228
fft, 228
flipud, 58
fminsearch, 50, 103
for, 17
fsolve, 67
function, 25
ga, 132
getframe, 166
gmres, 46, 51, 116, 234
grid off, 34
grid on, 34
hist, 506
ifft2, 228
ifftshift, 228
ifft, 228
if, 17
image, 382
imread, 58, 382
imresize, 58
imshow, 382
inexact_alm_rpca, 431
inline, 84
interp1, 101
iron, 289
isosurface, 163
legend, 31, 154
linprog, 123
linspace, 30, 155
load, 35
mean, 327
meshgrid, 155
minimize, 118
movie, 165
ode45, 191
pcolor, 156
plot, 29
polyfit, 99
polyval, 99
print, 35
quad, 84
randintrlv, 432
randn, 327, 383
rand, 327
rgb2gray, 58, 382
save, 35
shading interp, 156
slice, 164
spdiags, 232
spy, 232
std, 327
str, 35
subplot, 33
surfc, 155
surfl, 155
surf, 155
title, 154
trapz, 83, 84
triplequad, 84
uint8, 383
var, 327
view, 161
waterfall, 158
wcodemat, 501
xlabel, 31, 154
ylabel, 31

accuracy
  ode45, 185
  double precision, 180
  global error, 179
  local error, 179
  numerical scheme, 179
actions, 514, 515
Adams method, 175, 186
adaptive time-stepping, 184
adjoint, 56
agent, 514, 515
autoencoder, 451

backpropagation, 140
bagging optimized DMD, 439
Bayes's formula, 317
Bellman equation, 520
blind source separation, 464
BOP-DMD, 439
boundary value problems, 187, 202

central limit theorem, 328
CFL condition, 246, 249, 250, 261
Chebychev polynomials, 270, 283
Chebychev transform, 270, 286, 287
classification trees, 508
clustering, 484, 496, 509
compressive sensing, 430, 540
conditional probabilities, 315
continuous wavelet transform, 362, 378
Cooley–Tukey algorithm, 269
Cosine transform, 267
covariance, 326, 417
cross-validation, 138, 493
curve fitting, 90
cvx, 124
CWT, 362, 378

data assimilation, 557
data labels, 136, 485
decoder network, 458
decoder networks, 451
deep learning, 554
dendrogram, 491, 492
dendrograms, 490
diagonalization, 408, 418
differentiation, 67, 69, 80, 85
dimensionality reduction, 413, 414
discretization, 67, 69, 72, 75, 80, 85, 171, 187, 199, 203, 219, 241, 247, 299

DMD, 436
dynamic mode decomposition, 436

eigenfaces, 57
eigenfunctions, 192, 207, 525
eigenvalues, 51, 56, 183, 192, 207, 408, 505
eigenvectors, 51, 56, 192, 207, 408
empirical orthogonal functions, 414
environment, 514, 515
EOF, 414
Euler method, 172, 180, 251
expectation, 324
expectation-maximization algorithm (EM), 493
exponential time-steppers, 185
extrapolation, 144

face recognition, 56
Fast ICA, 480
fast Poisson solvers, 225, 236
feature engineering, 484
feature spaces, 484
feed-forward neural networks, 214
FFT, 226, 227, 267, 338
filtering, 342
finite differences, 219
finite elements, 298
foundation models, 574
Fourier transform, 267, 335
Fouriertransform, 225
Fredholm-alternative theorem, 56

Gábor transform, 353, 358, 372
Gaussian mixture models (GMM), 492
Gaussian mixtures, 492
generalization, 144
genetic algorithm, 128
GMM, 492
gradient descent, 140

Haar wavelet, 361
harmonic oscillator, 213
hierarchical clustering, 490
hierarchical time-stepping, 554
Hotelling transform, 414

ICA, 462
image
  compression, 378
  denoising, 387
  processing, 387
  recognition, 495
  separation, 465
independent component analysis, 462
initial value problems, 172
integration, 75
interpolation, 101, 144

Jacobian, 65

k-means, 489
Kalman
  EKF, 564
  extended filter, 564
  filter, 562, 563
  gain, 562
  Lorenz equations, 566
  vector EKF, 564
Karhunen–Loéve decomposition, 414
Koopman operators, 442

labeled data, 496
Lagrange coefficients, 96
Lagrange polynomial, 77
Lax–Wendroff scheme, 248, 252
LDA, 501
leap-frog scheme, 252
least-square fit, 90, 99
linear discrimination analysis, 501
linear operator, 207
linear programming, 119
linear stability, 207, 291
linear systems, 38
  Gauss–Seidel iteration, 45
  Gaussian elimination, 38
  gradient descent, 46
  iterative techniques, 43, 225, 234
  Jacobi iteration, 43
  LU decomposition, 40
  matrix transpose, 15, 56, 60
  quadratic form, 47
  strict diagonal dominance, 44
long-short term memory network, 458
low-rank approximations, 413
LSTM, 458

MacCormack scheme, 249
machine learning, 134, 495
Markov decision process, 518
Markov transition matrix, 518
matrix, 11
matrix powers, 55
Mexican Hat wavelet, 364
mixture models, 492–494
model discovery, 549
moments, 324
mother wavelet, 360
MRA, 367
multi-resolution analysis, 367

neural networks, 214, 451, 458, 508, 554
Newton–Cotes, 76
Newton–Cotes formulas, 76
Newton–Raphson method, 21, 64
nonlinear SVD, 451, 458
nonlinear systems, 64, 99, 202, 205
numerical quadrature, 75
numerical round-off, 73, 180
Nyquist theorem, 393

objective function, 108, 128, 135, 559
observations, 514, 515
operator splitting, 256
optimization, 108
  fminbnd, 114
  derivative methods, 114
  fminsearch, 50, 103
  fsolve, 67
  ga, 132
  minimize, 118
  genetic algorithm, 128
  golden section search, 110
  gradient descent, 114
  linear programming, 119
  Newton–Raphson, 21, 64
  objective function, 108, 128
  simplex method, 118, 124
  steepest descent, 114

optimization (Continued)
successive parabolic interpolation, 112
unconstrained optimization, 114
optimized DMD, 439

PCA, 60, 414, 418, 422, 497, 524, 540
permutation matrix, 42
POD, 414, 422, 524, 540
POD with symmetry, 533
Poisson equation, 220, 225
policy, 520
polynomial fit, 95
predictor–corrector methods, 178
principal components, 60, 414, 422, 497, 524, 540
probability theory, 311
proper orthogonal decomposition, 414, 422, 524, 540
pyDMD, 439

quadratic form, 559

radar, 342
random variables, 318
randomized linear algebra, 448
randomized SVD, 448
recurrent neural networks, 458, 544
reduced order modeling, 414
regression, 134
reinforcement learning, 135, 514
RL, 514
robust PCA, 540
Runge–Kutta method, 174, 187

Savitsky-Golay, 87
sensing, 544
shallow recurrent decoders, 458, 544
shooting method, 188
short-time Fourier Transform, 353
SHRED, 458, 544
simplex method, 118, 124
Simpson's rule, 77
SINDY, 549
Sine transform, 267
singular value decomposition, 403, 414, 465
solvability, 56
sonar, 342
sparse identification of nonlinear dynamics, 549
sparse matrices, 232, 237
sparse regression, 104, 549
sparse sensing, 430, 540
sparsity, 104
spectra, 207, 291
spectral filtering, 342
spectral methods, 266, 274, 277, 291
spline, 85, 97, 102
stability
linear stability, 207, 291
numerical scheme, 179, 181, 249
statistical decisions, 329, 484, 495, 501
statistical methods, 311, 501
statistical significance, 327
STFT, 353, 358, 372
strictly diagonal dominant, 44
supervised learning, 135, 495, 496
support vector machines, 508
SVD, 403, 414, 418, 465, 497
SVM, 508

Taylor series, 70, 174, 200
test data, 136
time–frequency analysis, 335
total reward, 519
training data, 136, 484
training labels, 136
training sets, 57, 505
transformers, 574
transpose, 15, 56, 60
trapezoidal rule, 77

unlabeled data, 484
unsupervised learning, 135, 136, 484, 496

value function, 519
variance, 324, 417
vector, 11
visualization, 152
von Neumann analysis, 250

wavelets, 360, 367, 378, 497
windowed Fourier transform, 353