Introduction to Engineering Numerical Methods

Johan Larsson

Department of Mechanical Engineering
University of Maryland

Second Edition

December, 2024

Contents

Preface

This book grew out of the course on fundamental numerical methods that I have been teaching at the University of Maryland for several years. While some of the students in this class will do their PhD in some area of computational science, the majority are either undergraduates or graduate students who focus their research on other topics and take the course to learn the fundamentals of numerical methods. Given this, my experience is that the material needs to be presented in a way that is intuitive to engineers, with less initial emphasis on the mathematics. I naturally use the course as a way to improve the students' abilities and sense of comfort with math and programming, but I try to lead with intuition and end with math rather than the other way around. Over the years, I have ended up developing my own course notes reflecting this style. After having been asked by many students for access to scans of my hand-written and only occasionally organized notes, I eventually took the time to write them down in LaTeX. The first edition reflected what I managed to type during the Fall 2022 semester. This second edition reflects what I added during the Fall 2024 semester; I hope to add some additional material and more examples in the coming years.

The book is aimed at beginners of the subject. More comprehensive treatment at a similar introductory level can be found in, for example, the book by Chapra & Canale (2015). A more rigorous and mathematical treatment for advanced students can be found in, for example, Heath (2018). While some linear algebra is used in this book, I have completely avoided any concept of numerical linear algebra; the book by Trefethen & Bau III (1997) is recommended. Finally, the books by Bendat & Piersol (2010) and Sivia (2006) provide comprehensive coverage of the aspects of statistics that we only barely touch upon in this book; the book by Sivia (2006) is a particularly joyful read.

Johan Larsson
Rockville, MD
December, 2024

Chapter 1

Introduction

Most readers of this book will have spent at least 15 years in school, with mathematics having been a recurring subject for most of that time. In math classes, students will have been taught to compute quantities, solve algebraic and differential equations, perform integration and differentiation, and many other things. So if readers have learnt so much math, why should they learn about numerical methods? Because the vast majority of real-world problems can't be solved with type of math that you have spent 15 or more years learning! Think about it: at some point you learnt how to solve equations of form $a + bx = 0$, and then later you learned to solve the more complicated $a + bx + cx^2 = 0$. But what about solving $a + bx + cx^2 + \ldots + ex^5 = 0$, or $a + bx^2 + \sin(x) + \ln(x) = 0$ – surely such equations might occur in real-world problems?

The purpose of numerical methods is to provide ways for us to solve whatever equations we want, to perform integrals of whatever functions we want, and so on. This is done by creating algorithms that are executed on computers. An algorithm is effectively a recipe, a step-by-step codified process which, if followed, leads to the final answer. Many algorithms require a lot of repeated steps – hence the need for the computer.

The price we pay for the ability to solve whatever equations we want comes in two parts: (i) we need to use computer resources, which costs both money and energy; and (ii) we must be satisfied with approximate answers. A user of numerical methods must have a basic level of understanding of both of these drawbacks in order to choose the right method for each problem context.

Example 1.1: the Babylonian method for computing square-roots

The Babylonians supposedly invented the following algorithm for computing $x = \sqrt{a}$ over 3000 years ago:

1. guess a value for x, and call this x_0
2. compute $x_1 = (x_0 + a/x_0)/2$
3. compute $x_2 = (x_1 + a/x_1)/2$
4. compute $x_3 = \ldots$, and keep going until the sequence of x_i values stops changing

To see how this algorithm works, let's choose $a = 100$. The correct ("analytical") answer is then $x = 10$, but if we don't know that we might choose our initial guess to be

$x_0 = 50$, say. In that case, the Babylonian algorithm produces: $x_1 = 26$, $x_2 = 14.9\ldots$, $x_3 = 10.8\ldots$, $x_4 = 10.03\ldots$, $x_5 = 10.00005\ldots$, $x_6 = 10.0000000001\ldots$, and so on. The algorithm clearly converges to the right answer, and thus provides us with a way to make this computation even if we don't know how to compute a square-root. After 6 iterations the error is about 10^{-10}, and the error is going down quickly with each additional iteration. However, it will *never* reach zero, and thus we must stop this process at some point and accept that the answer is not exact.

1.1 Numerical methods vs numerical analysis

The topic of numerical methods can be approached from two rather different perspectives. For lack of better descriptors, let's call these approaches "numerical methods" and "numerical analysis".

- **The numerical methods perspective:** On one extreme end, a person can find an algorithm online or in a book, write some code that implements this algorithm in some programming languange, and then use this to solve the problem at hand. This approach is analogous to cooking food: you read a recipe and then do your best to follow that recipe, without needing to know anything about the chemistry happening during the cooking process. This approach to the subject requires a degree of "just do it" attitude, and a willingness to use trial-and-error just to see what happens.
- **The numerical analysis perspective:** A very different approach to the subject is the more careful and analytical one, in which one might say "if the answer provided by this algorithm is approximate, surely it would be useful to know how large the error is" or "if I will use computer power for this, surely it would be useful to know how much computing power will be needed – can I use my cell phone, my laptop, or do I need a supercomputer?". One can then use the tools of math to analyze the algorithm itself: i.e., rather than use analytical math to solve the problem, one would use those same math tools to analyze the algorithm. This may seem strange at this point, but hopefully it will become clear throughout the book just how useful this can be.

When first studying this topic, the distinction between numerical "methods" (implementing a method) and "analysis" (using math to understand how a method will behave) will probably seem fuzzy. For that reason, this book tries to make a distinction between these perspectives when covering different types of methods.

1.2 Different types of errors

Numerical methods are (almost) never able to compute the exact answer, and therefore it is reasonable to ask what kind of errors we might encounter. In principle there are many kinds of errors, although not every numerical method will necessarily be affected by every single kind.

The Babylonian method for computing square-roots (see the example above) is affected by two different kinds of errors. The most obvious one is that the algorithm must be terminated at some point, meaning that we will in reality compute a finite number of iterations. Thus an important part of this method (or "recipe") is the clear specification of how one should decide when to terminate. Understanding when to terminate requires an understanding of how the error in x_i is decreased at each iteration, something which can be estimated through numerical analysis.

The second type of error affects all numerical methods, in fact it affects every algorithm that is implemented on a computer. It's called "round-off" error, and is related to the fact that a computer stores real numbers with finite precision. In "single precision", a computer uses 4 bytes (=32 bits, or 32 zeros or ones) to store real numbers as $\pm\,\text{significand}\cdot\text{base}^{\text{exponent}}$, with 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand (the base is 2). In "double precision", the exponent gets 11 bits and the significand gets 52 bits. This works out to precisions of about 10^{-7} and 10^{-16}, respectively, when measured in a relative sense.

The round-off error is a fundamental limit of using a computer. In the best case scenario our final answer will thus have a relative error of 10^{-16} (assuming double precision), but since the round-off error affects pretty much every single step in a calculation, there is a risk that all of the round-off errors build up to a larger relative magnitude in the final answer. Understanding how robust an algorithm is to round-off error is therefore quite important.

Example 1.2: round-off error

Compute $\sqrt{2}^2 - 2$ using some computing language of your choice (Matlab, Python, C++, ...). Make sure you do it in steps: first $\sqrt{2}$, then square the result, then subtract 2. If you do this in double precision, you'll find that the result is $4.44\cdot 10^{-16}$, rather than the value of zero that it should be; this is the round-off error stemming from storing the result of the square-root operation in double precision.

1.3 Required background for this book

The following contains very brief reviews of some mathematical concepts that are used throughout this book. Readers who are unfamiliar or uncomfortable with this material should consult an undergraduate mathematics text book.

1.3.1 Taylor expansions

Consider a function $f(x)$ that has continuous derivatives up to order $n+1$ between x_0 and x. In that case, we can write

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \ldots + f^{(n)}(x_0)\frac{(x-x_0)^n}{n!} + R_{n+1}\,, \tag{1.1}$$

with the remainder term

$$R_{n+1} = f^{(n+1)}(t)\frac{(x-x_0)^{n+1}}{(n+1)!}$$

where t is between x_0 and x (in math notation, $t \in [x_0, x]$).

Note that we made use of the short-hand notation $f'(s) = df/dx|_s$, $f''(s) = d^2f/dx^2|_s$, etc, and, for the nth derivative, $f^{(n)}(s) = df^n/dx^n|_s$.

A slightly different way to write the Taylor expansion (1.1) is to replace the remainder term R_{n+1} with the less precise

$$\mathcal{O}\left((x-x_0)^{n+1}\right)\,,$$

which should be interpreted as a term of unknown exact value but which varies as $(x-x_0)^{n+1}$ if $(x-x_0)$ is varied. This notation is used heavily in this book.

Taylor expansions are very commonly used in numerical analysis. In an extremely hand-wavy way, the Taylor expansion (or Taylor series) in Eqn. (1.1) says that we can either describe $f(x)$ by knowing the function values at all different locations x (the left-hand-side of the equation) or we can equivalently describe $f(x)$ by knowing all different derivative values at a single location x_0 (the right-hand-side). An example is shown in Fig. 1.1.

1.3.2 Complex variables

A complex-valued variable z can be written in multiple forms, including

$$z = z_r + \imath z_i = |z|e^{\imath\theta}\,,$$

where $\imath = \sqrt{-1}$ is the imaginary unit, z_r is the real part, z_i is the imaginary part, $|z|$ is the magnitude, and θ is the angle of the complex number. Addition and subtraction are most easily performed using the first form, while multiplication and division are most easily performed using the second form. An important identity is that

$$e^{\imath\theta} = \cos(\theta) + \imath\sin(\theta)\,.$$

This can then be manipulated into several useful formulae, including

$$\begin{aligned}\cos(\theta) &= \frac{e^{\imath\theta} + e^{-\imath\theta}}{2}\,,\\ \sin(\theta) &= \frac{e^{\imath\theta} - e^{-\imath\theta}}{2\imath}\,.\end{aligned}$$

The complex conjugate z^* of a number z is defined as having the same magnitude but the opposite (negative) angle, i.e., $z^* = |z|e^{-\imath\theta} = z_r - \imath z_i$.

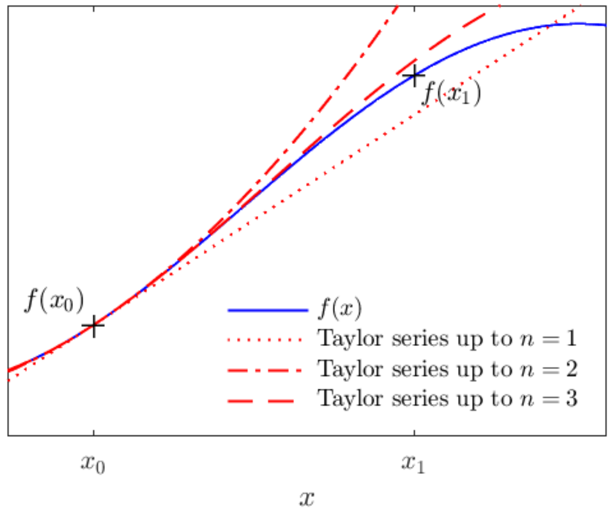

Figure 1.1: Illustration of the Taylor expansions, approximating a function $f(x)$ around the base point x_0 using two, three and four terms (including the $f(x_0)$ term).

1.3.3 Linear algebra

Consider an n-dimensional vector

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},$$

which we could also write more compactly as $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ where the superscript T denotes the transpose.

An $m \times n$ matrix is defined as

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},$$

with the components a_{ij} defined with the first index being the row and the second being the column.

The matrix-vector and matrix-matrix products are defined only when the two factors are compatible in size, specifically the number of columns in the first must equal the number of rows in the second. The product

$$A = BC$$

is most easily defined using index notation of the components; specifically, the product is

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj} \,.$$

The same expression holds if C is a vector, in which case $j = 1$ is the only column of C and thus A. Similarly, if B is a row vector, then $i = 1$ is the only row of B and thus A.

A linear system of equations is written as

$$A\mathbf{x} = \mathbf{q} \,.$$

If A is a square non-singular matrix, the solution to this system is $\mathbf{x} = A^{-1}\mathbf{q}$ where A^{-1} is the inverse of A. If A has more rows than columns, the system is over-determined since there are more equations (=rows in A) than unknowns (=columns in A). If A has fewer rows than columns, the system is under-determined.

The matrix A is called "sparse" if it has mostly zeros. Having a sparse matrix is very advantageous in numerical linear algebra since it may drastically reduce the computational effort required to find the inverse. The basic way to find the inverse (or solve a linear system of equations) is to use Gaussian elimination, which has a computational complexity (=cost) that is proportional to n^3 for an $n \times n$ matrix. In contrast, a tri- or penta-diagonal matrix (i.e., a matrix with non-zero elements only on the main diagonal and on one or two diagonals on either side) can be solved using the Thomas algorithm at a computational complexity that is proportional to only n!

A square $n \times n$ matrix A has n eigenvectors $\mathbf{x}_i$ and n eigenvalues λ_i defined by

$$A\mathbf{x}_i = \lambda_i \mathbf{x}_i \ , \quad i = 1, 2, \ldots, n \,.$$

This relation can also be written as

$$AX = X\Lambda \,,$$

where X is a matrix where each column is one of the eigenvectors, i.e.,

$$X = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \ldots & \mathbf{x}_n \end{pmatrix}$$

and Λ is a diagonal matrix with the eigenvalues on its diagonal, i.e.,

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & \ldots & \\ 0 & \lambda_2 & 0 & \\ & 0 & \ddots & 0 \\ & \ldots & 0 & \lambda_n \end{pmatrix} \,.$$

1.3.4 Statistics

Consider a random variable X with expected value $E(X) = \mu_X$ and variance $V(X) = \sigma_X^2$. If we have a set of random samples $x_i, i = 1, \ldots, n$, we can compute the sample mean

$$\overline{x} = \frac{1}{n} \sum_{i=1} x_i \,,$$

and sample variance

$$s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 \,.$$

Since the set of x_i are random samples, by implication the sample mean $\overline{x}$ and sample variance s_x^2 are random too; in other words, if we repeat the experiment, we get a new set of values x_i and thus new computed sample mean and sample variance. In contrast, the expected value μ_X and variance σ_X^2 are *not* random, but instead deterministic attributes of the random variable.

The purpose of most data collection is to use the collected random samples to say something about the underlying random variable; this process is called inference. A basic tool in statistical inference is the confidence interval. If applied to the expected value, a simple confidence interval is

$$\mu_X = \overline{x} \pm z_{\alpha/2}\,\sigma_{\overline{x}}\,, \tag{1.2}$$

which is an abbreviated way of saying that the expected value of X (the "true mean") differs from the computed sample mean $\overline{x}$ by less than the amount $z_{\alpha/2}\,\sigma_{\overline{x}}$ with $1-\alpha$ confidence or probability. The value $z_{\alpha/2}$ provides the link between the confidence level and the size of the interval: larger confidence requires a larger interval. The quantity $\sigma_{\overline{x}}$ is the so-called *standard error*, which quantifies the variability in the computed sample mean. For uncorrelated (or independent) samples, the standard error is $\sigma_X/\sqrt{n}$, which produces the formula for a confidence interval (for the expected value) that is covered in most undergraduate statistics courses.

Part I

Basic building blocks

Chapter 2

Root-finding

Consider the function $f(x)$ and the equation $f(x) = 0$. The values x_* for which $f(x_*) = 0$ are called the "roots" of the equation. In general there can be many roots of an equation, perhaps even infinitely many (e.g., the equation $\sin(x) = 0$). "Root-finding" methods attempt to find a single root from either an initial guess x_0 or a specified interval $[x_0, x_1]$. In order to find multiple roots, a user would then have to try different initial guesses or different specified intervals.

There are many root-finding methods, some of which will be presented here. Some methods work well on some problems, others work better on other problems – one must therefore know multiple root-finding methods (and their strengths and weaknesses) in order to be able to handle a wide range of problems.

2.1 Fixed-point iteration

The perhaps simplest root-finding method is the fixed-point iteration method, in which one re-writes the original equation $f(x) = 0$ to $x = g(x)$ and solves by first choosing an initial guess x_0 and then updating this initial guess as

$$x_{i+1} = g(x_i)\,, \tag{2.1}$$

where i is the iteration counter. For example, if $f(x) = x^2 + x + 1$, a simple choice would be $g(x) = -x^2 - 1$, since it is easy to see that $x = g(x)$ and $f(x) = 0$ are then the same equations. However, one could also choose $g(x) = x^2 + 2x + 1$, which hints at the fact that there is an element of user intuition required in this method. More general choices include $g(x) = x + \alpha f(x)$ or $g(x) = \alpha/(\alpha/x + f(x))$, just to provide some additional examples; in both cases, provided $\alpha \neq 0$, a fixed point $x = g(x)$ then also implies $f(x) = 0$.

The choice of $g(x)$ has a direct impact on whether the method converges or not, and how fast it converges. Convergence of a root-finding method is defined as having the iterates $x_i \to x_*$ as $i \to \infty$. If the difference between successive iterates (x_i and x_{i+1}, say) becomes smaller for every iteration, the convergence is said to be monotonic. In practice, one should not expect monotonic convergence during the initial iterations when x_i is far from the root x_*, but one would hope for monotonic convergence once

x_i is getting close to x_*. If successive iterates are identical, i.e., if $x_i = x_{i+1}$, then they solve the equation $x = g(x)$ and thus, by construction, also the original equation $f(x) = 0$: in other words, we would have found the exact root x_*. The name of the method comes from the fact that successive iterates are identical, or a "fixed point" of the iterative map.

Having described the algorithm for fixed-point iteration, or the "recipe" for how to do it, let us now analyze this algorithm. It is important to remember that this analysis phase has nothing to do with how to implement the method in a code, it is only a theoretical tool for understanding how the method works, when it might fail or not, and so on.

For x values close to the true root x_*, a Taylor expansion yields

$$g(x) \approx \underbrace{g(x_*)}_{=x_*} + g'(x_*)(x - x_*) + \mathcal{O}\big((x - x_*)^2\big) \ .$$

Defining the error $e_i = x_i - x_*$, the algorithm (2.1) then implies that

$$e_{i+1} \approx g'(x_*)e_i + \mathcal{O}\big(e_i^2\big) \ .$$

Convergence implies that the error becomes smaller in magnitude during each iteration, which means that convergence requires $|g'(x_*)| < 1$. This requirement is then useful when deciding how to define $g(x)$, as the different possibilities mentioned above would have different slopes $g'(x_*)$ at the root. There is no single way to choose $g(x)$ that leads to convergence for all functions $f(x)$, and thus this fixed-point iteration approach is ill-suited for general purposes; having said that, it can be powerful for certain equations.

2.2 Bisection method

The bisection method is arguably the most common root-finding method, and chances are that most readers of this book have already been exposed to it. The method is based on the following observation: if $f(x)$ is real-valued and continuous on the interval $[x_l, x_u]$ (meaning, for $x_l \leq x \leq x_u$), and if $f(x_r)$ and $f(x_u)$ have opposite signs, then the function must have crossed zero at some point in that interval, and thus there must exist a root $x_* \in [x_l, x_u]$. Given this, the essence of the method is then to first choose a new "trial" point x_t somewhere in the interval, evaluate $f(x_t)$, and finally to check the sign of the function value: if the function has the same sign at the trial point x_t as at the lower (or left) boundary x_l, then we know that the root must lie between x_t and x_u and we can replace the lower bound with x_t instead. If the function has the same sign at x_t as at x_u, then we would replace the upper (or right) bound x_u instead. In either case, our new interval (either $[x_t, x_u]$ or $[x_l, x_t]$) is smaller than before, which means that we have more knowledge of where the root is.

The description above did not specify how to choose the new trial point, only that it must be inside the interval. The bisection method takes its name from the choice of x_t as exactly halfway between the interval bounds: i.e., the interval is bisected by the trial point. In this way, the length of the interval is halved at each iteration. As will

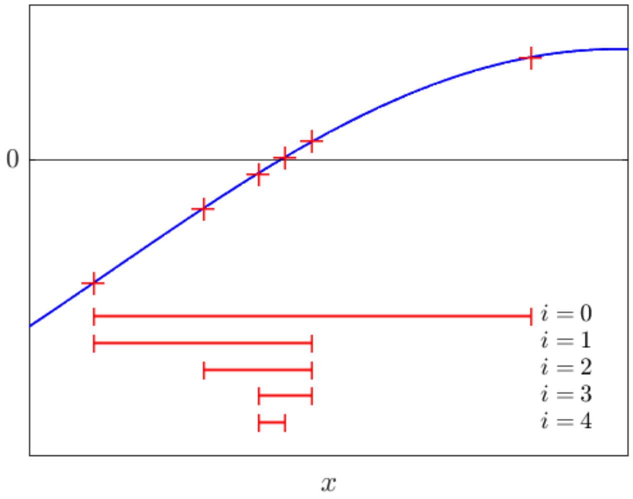

Figure 2.1: Sketch of the bisection root-finding method, showing how the interval (horizontal red bars) shrinks with each iteration.

discussed below, one could make other choices, but then it's not strictly a "bisection" method.

The algorithm is then

Algorithm 2.1: the bisection method

```
guess an initial interval $[x_l, x_u]$
while not converged,
   compute trial point $x_t = (x_l + x_u)/2$
   if $f(x_t)f(x_l) > 0$,        (i.e., if the function has the same sign at $x_l$ as at $x_t$)
      $x_t \leftarrow x_l$              (replace $x_l$ by $x_t$)
   else,
      $x_t \leftarrow x_r$              (replace $x_u$ by $x_t$)
   end if
end while
take the final estimated root as $(x_l + x_u)/2$
```

with a sketch of the method shown in Fig. 2.1.

What remains in terms of specifying the algorithm is the question of when to stop the process, i.e., how to decide when the estimated root is sufficiently close to the true root x_*. Since the true root is unknown (otherwise there would be no need for a root-finding algorithm!), the best we can do is make an estimate of the error. Two immediate possibilities are $x_u - x_l$ (the size of the interval) and $|f(x_t)|$ (the mismatch in the equation $f(x) = 0$). In both cases, a user would need to specify an acceptable tolerance for their specific case. In most situations, it is more natural to specify this tolerance in the context of the error in the root itself than in the function value (or equation mismatch); thus the most natural stopping criterion is framed in terms of $x_u - x_l$.

Having described the method and the algorithm, we next turn to some analysis. The error at iteration i is

$$e_i = x_{t,i} - x_* = \frac{x_{l,i} + x_{u,i}}{2} - x_* \,.$$

We don't know the true root x_*, but we know that the error is bounded in magnitude as

$$|e_i| \leq \frac{x_{u,i} - x_{l,i}}{2} \,,$$

i.e., the error can not be larger in magnitude than half the length of the interval. Since the interval is cut in half in each iteration, this implies that

$$|e_i| \leq \frac{x_{u,0} - x_{l,0}}{2} \left(\frac{1}{2}\right)^i \quad , \quad i = 0, 1, \ldots \,,$$

where subscript 0 means the initial guess. This is a quite powerful statement: it's a hard error bound that depends only on the choice of initial interval and the number of iterations. This property of the bisection method thus implies that convergence to whatever tolerance one desires is guaranteed, provided one chooses an initial interval that contains the root (and that $f(x)$ is continuous).

2.3 Newton-Raphson method

The Newton-Raphson method is a potentially very fast root-finding method, and the derivation of it shows how numerical *analysis* can be used to derive a numerical *method.* As before, we want to find the root x_* that solves $f(x) = 0$. Assuming that we have a guess x_i that is close to the root, the idea is to create a linear approximation of $f(x)$ around x_i and then to find where this linear approximation crosses zero, which is then taken as the updated value x_{i+1}. The linear approximation of $f(x)$ is taken as

$$f_{\text{linear}}(x) = f(x_i) + f'(x_i)(x - x_i) \,,$$

i.e., as the first two terms of a Taylor expansion around x_i. This guarantees that the linear approximation $f_{\text{linear}}(x)$ and $f(x)$ have exactly the same function values and derivatives at x_i. Finding the root (= zero-crossing) of this linear approximation is done by setting $f_{\text{linear}}(x) = 0$; labeling the root x_{i+1} then yields

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \,, \tag{2.2}$$

which is the Newton-Raphson method. A sketch of the method is shown in Fig. 2.1.

To implement this method, one needs an initial guess x_0 and the ability to compute both $f(x)$ *and* $f'(x)$ for any given x. The need for $f'(x)$ limits the applicability of this method. In some scenarios it is easy to find $f'(x)$ analytically (it is often easier to take

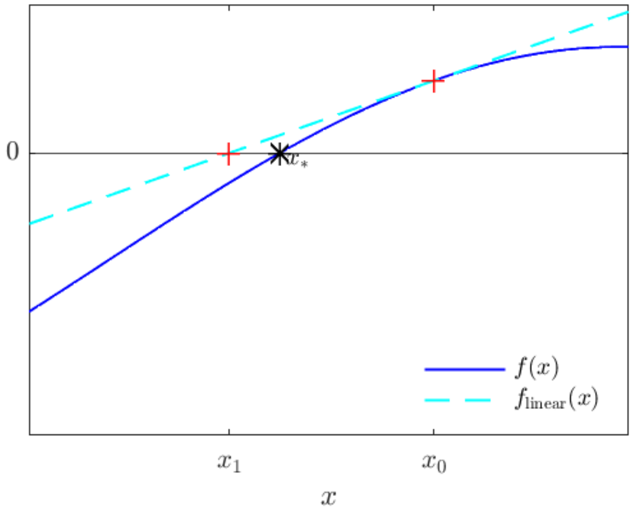

Figure 2.2: Sketch of the Newton-Raphson root-finding method.

a derivative than to find a root), but in others this may not be easy or possible at all, in which case the Newton-Raphson method can not be used.

Having described the numerical method, we next turn to some numerical analysis. As before, this is centered on the behavior of the error $e_i = x_i - x_*$, and as before we use Taylor expansions to estimate how the error behaves when x_i is close to x_*. Specifically, we have

$$e_{i+1} = x_{i+1} - x_* = x_i - \frac{f(x_i)}{f'(x_i)} - x_* = e_i - \frac{f(x_i)}{f'(x_i)}\,, \tag{2.3}$$

which relates the errors at two successive iterations. Clearly there is no bound on this error due to the explicit dependence on $f(x_i)$ and $f'(x_i)$. As a result, we can *not* prove convergence for this method: unlike the bisection method, it is entirely possible that the Newton-Raphson method diverges (i.e., fails to find the root) for some functions $f(x)$.

Even if we can't say anything about whether the method will converge, can we say anything useful at all? Actually, yes we can. Assuming that x_i is close to x_*, let us Taylor-expand both $f(x_i)$ and $f'(x_i)$ in the error equation (2.3) around x_*; this yields (recall that $e_i = x_i - x_*$, and that $f'(x)$ can be Taylor-expanded just like any other function)

$$e_{i+1} \approx e_i - \frac{f(x_*) + f'(x_*)e_i + f''(x_*)e_i^2/2 + \mathcal{O}\big(e_i^3\big)}{f'(x_*) + f''(x_*)e_i + f'''(x_*)e_i^2/2 + \mathcal{O}\big(e_i^3\big)}\,. \tag{2.4}$$

The goal of this analysis is to see how the error changes during one iteration: in order to see that, we next need to Taylor-expand the denominator. To do this, we can define

$$g(e_i) = \frac{1}{f'(x_*) + f''(x_*)e_i + f'''(x_*)e_i^2/2}$$

and find the derivative

$$g'(e_i) = -\frac{1}{\left[f'(x_*) + f''(x_*)e_i + f'''(x_*)e_i^2/2\right]^2}\ \left[f''(x_*) + f'''(x_*)e_i\right]$$

and finally Taylor-expand $g(e_i)$ as

$$\begin{aligned} g(e_i) &\approx g(0) + g'(0)e_i + \mathcal{O}\big(e_i^2\big) \\ &= \frac{1}{f'(x_*)} - \frac{f''(x_*)}{[f'(x_*)]^2}\,e_i + \mathcal{O}\big(e_i^2\big) \\ &= \frac{1}{f'(x_*)}\left[1 - \frac{f''(x_*)}{f'(x_*)}\,e_i + \mathcal{O}\big(e_i^2\big)\right] . \end{aligned}$$

Inserting this into the error equation (2.4) then yields, also using the fact that by definition $f(x_*) = 0$,

$$\begin{aligned} e_{i+1} &\approx e_i - \left[f'(x_*)e_i + \frac{f''(x_*)}{2}e_i^2 + \mathcal{O}\big(e_i^3\big)\right]\frac{1}{f'(x_*)}\left[1 - \frac{f''(x_*)}{f'(x_*)}\,e_i + \mathcal{O}\big(e_i^2\big)\right] \\ &\approx e_i - \left[e_i + \frac{f''(x_*)}{2f'(x_*)}e_i^2 + \mathcal{O}\big(e_i^3\big)\right]\left[1 - \frac{f''(x_*)}{f'(x_*)}\,e_i + \mathcal{O}\big(e_i^2\big)\right] \\ &\approx \frac{f''(x_*)}{2f'(x_*)}\,e_i^2 + \mathcal{O}\big(e_i^3\big) . \end{aligned}$$

So, when the error is small (i.e., when we are close to the root), the magnitude of the error in Newton-Raphson behaves as

$$|e_{i+1}| \approx \left|\frac{f''(x_*)}{2f'(x_*)}\right| e_i^2 .$$

In other words, the error is reduced quadratically – if the root is known to 4 significant digits after iteration i, it is known to 8 significant digits after iteration $i+1$. That is quite remarkable.

To summarize, our analysis of the Newton-Raphson method has concluded that it may not converge in all cases, but when it does, it converges extremely fast.

2.4 Secant method

The Newton-Raphson requires the ability to compute the derivative $f'(x)$ which can be difficult to implement in some situations. For example, imagine that x is the angle-of-attack of an aircraft and $f(x)$ is the lift force created by the wing, and that we are trying to find the angle-of-attack x_* for which the lift force is zero. So this is a root-finding problem, but the evaluation of $f(x)$ may require the running of a big simulation code on a supercomputer, or it may involve running an experiment in a wind tunnel. In either case, while we have the ability to evaluate $f(x)$ (albeit at a very high cost), we do not have the ability to evaluate $f'(x)$.

The secant method is essentially an approximate version of Newton-Raphson where one uses the last two iterates x_i and x_{i-1} to approximate the derivative as

$$f'_{\text{approx}}(x_i, x_{i-1}) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}\,.$$

The secant method is then derived from the Newton-Raphson method (2.2) as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{\text{approx}}(x_i, x_{i-1})} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}\,.$$

It will obviously require two initial guesses from the user.

The analysis of the secant method is a bit difficult since it involves three different iteration levels, and we therefore skip this part in this introductory book. The outcome of the analysis is that the secant method converges a bit more slowly than the Newton-Raphson method but still faster than linearly; it can therefore be a powerful method in many scenarios.

Example 2.1: two root-finding problems

Consider the two root-finding problems

$$\begin{aligned} f_1(x) &= 2\sin(x) - 1 = 0\,, \\ f_2(x) &= (x-1)^3 = 0\,. \end{aligned}$$

The exact roots are $x_{*,1} = \pi/6$ and $x_{*,2} = 1$, but let us instead compute them using the methods described here. We take the initial guess $x_0 = 0$ for the Newton-Raphson method. We take the additional initial guess $x_1 = 0.2$ for the secant method, and the interval of $[0, 1.8]$ for the bisection method (we want to avoid finding the exact root by luck). The functions and the resulting convergence behavior (the absolute value of the error in the estimated root at each iteration) are shown in Fig. 2.3.

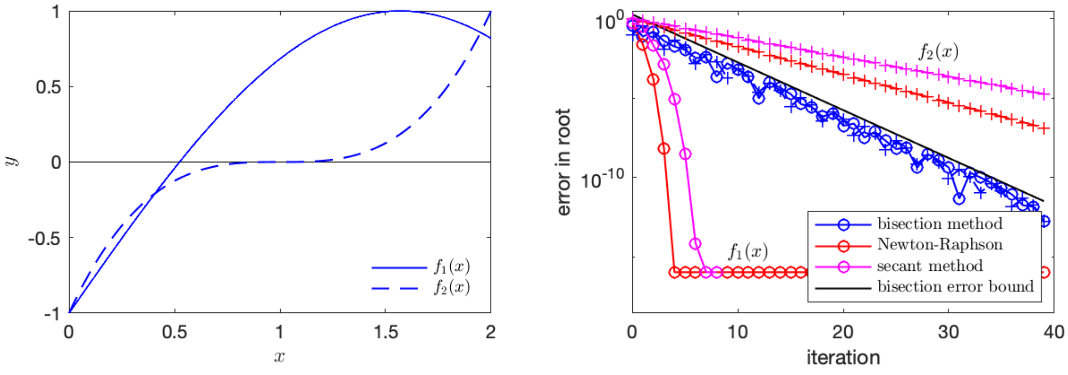

Figure 2.3: Root-finding algorithms applied to two different problems $f_1(x) = 0$ and $f_2(x) = 0$) (left) and the resulting convergence behavior (right).

The bisection method produces the same convergence behavior for both problems: this method is “slow” but always converges (provided the function is continuous and the initial interval is valid). The figure also shows the error bound, and it is clear that the actual error is always below this bound.

The Newton-Raphson method produces incredibly fast convergence for $f_1(x) = 0$ but very slow convergence for $f_2(x) = 0$. In fact, the convergence is linear for the second problem: this is caused by this problem having a multiple (or repeated) root.

The secant method behaves qualitatively like the Newton-Raphson method, but not quite as efficiently; this slight lack of convergence speed is generally a small price to pay for avoiding the need to evaluate the derivative.

Chapter 3

Interpolation

The general interpolation problem in one dimension can be stated as: assuming that you have a set of N data points (x_i, y_i) (numbered $i = 1, \ldots, N$), what is the estimated value of y for some different value of x?

Within this general problem statement there are several different flavors of interpolation methods. One key distinction is whether one wants the interpolation function to pass through the data points or not; this is almost equivalent to asking whether the data are affected by random noise or not. This is illustrated in Fig. 3.1. If the 4 data points are viewed as being exact, then they suggest that the true function varies non-monotonically, and the red interpolating curve might be a reasonable choice. If, on the other hand, one believes that the true function should be monotonic in this interval and therefore that the variation in the data points must come at least partly from noise, then the green interpolating function might be a more reasonable choice. This chapter will cover both types of interpolation strategies.

Interpolation methods also differ in whether a single interpolating function is used everywhere or not. Two of the examples in Fig. 3.1 use a single ("global") interpolating function, whereas the piecewise linear interpolation method actually uses $N-1$ different interpolating functions depending on where you want to perform the interpolation; in a sense, this method is "local".

Finally, interpolation methods also differ in what types of interpolating functions are used. All interpolating functions in Fig. 3.1 are polynomials, but there are other choices. The most common different choice is to use trigonometric functions (sines and cosines) as interpolating functions, which might make sense if one knows that the true function is periodic (or nearly so). If one allows for the interpolating function to be a sum of sines and cosines, then this becomes a Fourier series.

3.1 Piecewise linear

The most straightforward interpolation strategy is to assume that the true function varies linearly between data points, and that it goes exactly through the data points. This is called piecewise linear interpolation, where the piecewise refers to the fact that we are actually using $N-1$ different interpolating functions in the $N-1$ different intervals

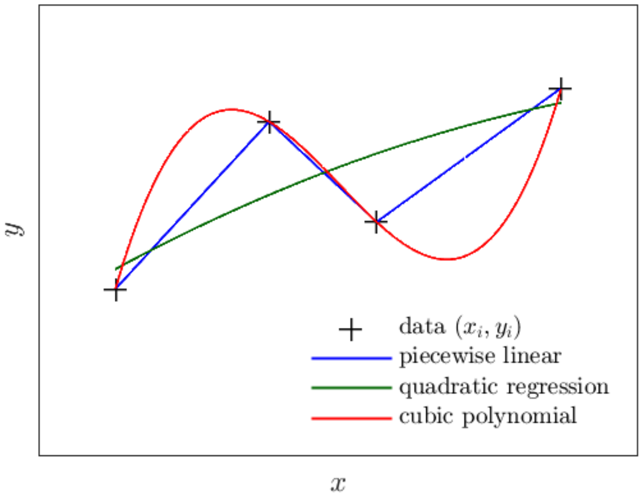

Figure 3.1: General interpolation problem, with given data points (x_i, y_i) and different possible interpolating functions.

between the N data points. The algorithm is necessarily composed of two parts: (1) we must first identify which interval our point x is in (i.e., which interpolating function to use); and then (2) we can perform a linear interpolation in that interval.

To find the interval, it is necessary that the data points are ordered, meaning that $x_1 < x_2 < x_3$ and so on. We can then define interval i as being the interval between x_i and x_{i+1}. If our data points are numbered $i = 1, 2, \ldots, N$, then our intervals are numbered $i = 1, 2, \ldots, N-1$.

Assuming that our point x for which we want to know the function value $y(x)$ is located in interval i, i.e., that $x \in [x_i, x_{i+1}]$. The interpolating function is then

$$f_i(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i} (x - x_i) \ ,$$

where we have used a subscript i for the interpolating function $f_i(x)$ to show that it applies only to the ith interval. It is sometimes convenient to use different (but entirely equivalent) forms of the interpolating function; it is quite straightforward to see that we can also write

$$f_i(x) = \left(1 - \frac{x - x_i}{x_{i+1} - x_i}\right) y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1}$$

or

$$f_i(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i} y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1} \, . \tag{3.1}$$

Both of these alternate forms highlight the fact that linear interpolation is effectively a weighted average between the two data points y_i and y_{i+1}, with the weight given by the scaled distance from the *other* point (i.e., the weight for y_i is the distance from x

to x_{i+1}, and the weight for y_{i+1} is the distance from x to x_i, scaled by the distance between the two points).

The main advantages of piecewise linear interpolation are the simplicity and the robustness. The fully "local" nature of the method implies that the interpolation is completely unaffeced by data points other than the two defining the interval, which makes the interpolation method very robust. The main disadvantage is the fact that the interpolating function is only C^0: it is continuous but not continuously differentiable.

3.2 Global polynomials

We next consider a single interpolating polynomial that is assumed valid over the full range of the data. Such a polynomial can be written as

$$f(x;\mathbf{a}) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \,, \tag{3.2}$$

where $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_n)^T$ is the vector of the $n+1$ polynomial coefficients. Interpolation using this polynomial $f(x;\mathbf{a})$ then has to proceed in two steps: (1) we must first find the coefficients $\mathbf{a}$ from the data points; and (2) we can then perform the interpolation by evaluating the polynomial.

We will use the polynomial in Eqn. (3.2) for interpolation both when the interpolating function goes through the data points and when it does not (i.e., regression). Assume at first that we want the polynomial to exactly agree with the data in the data points. This amounts to requiring that, for every data point $i = 1, 2, \ldots, N$,

$$f(x_i;\mathbf{a}) = a_0 + a_1 x_i + a_2 x_i^2 + \ldots + a_n x_i^n = y_i \ , \quad i = 1, 2, \ldots, N \,. \tag{3.3}$$

This amounts to N equations for $n+1$ unknown coefficients (recall that x_i and y_i are known). We can write this system of equations in matrix-vector form as

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^n \\ 1 & x_2 & x_2^2 & \ldots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_i & x_i^2 & \ldots & x_i^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \ldots & x_N^n \end{pmatrix}}_{A} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{pmatrix}$$

or $A\mathbf{a} = \mathbf{y}$.

It is useful to consider under what conditions this system of equations can be solved. We first note that the matrix A is a so-called Vandermonde matrix, with rows that are linearly independent if and only if all the data locations x_i are unique (i.e., if there are two data points with the same x value, then A has two rows that are the same). We then have the following possible scenarios:

- If $N = n+1$ and if all the data locations x_i are unique, then A is invertible and we can find a unique set of polynomial coefficients $\mathbf{a}$. For example, if we have $N = 3$ data points we can exactly fit a unique quadratic ($n = 2$) polynomial.

- If $N < n+1$, then we have an underdetermined system which means that we can find infinitely many solutions $\mathbf{a}$. For example, if we have $N = 2$ data points, then we can find infinitely many different parabolas ($n = 2$) that go through those two data points. This is not very useful.
- If $N > n+1$, then we have an overdetermined system which means that we almost certainly can not find a solution $\mathbf{a}$. In English, this means that we can't find a parabola that goes through $N \geq 4$ different data points, or a straight line that goes through $N \geq 3$ different data points (unless the data lines up perfectly, of course). However, while we can't find a polynomial that exactly agrees with the data, we can find a polynomial that approximately agrees with the data – this is called regression in the field of statistics.

So, given N data points, we have two choices for polynomial interpolation: we can either (1) find the polynomial of order $n = N - 1$ that exactly goes through the data (by solving $\mathbf{a} = A^{-1}\mathbf{y}$); or (2) find a polynomial of any order $n < N - 1$ that approximately goes through the data. In the latter case, we must choose which polynomial order n we want.

3.2.1 Approximate agreement with the data

The big question is how to find the coefficients $\mathbf{a}$ when we don't want the interpolating function to exactly agree with the data. Recall that our equation $A\mathbf{a} = \mathbf{y}$ comes from requiring exact agreement with the data points – if we don't want to enforce that, what should we enforce? The best way to start is to replace Eqn. (3.3), which enforces exact agreement, with

$$f(x_i; \mathbf{a}) = y_i + e_i \ , \quad i = 1, 2, \ldots, N$$

or in matrix-vector form

$$A\mathbf{a} = \mathbf{y} + \mathbf{e} \, ,$$

where we have introduced the misfit (or "residual") $\mathbf{e}$. With this misfit, the equation is again valid. However, this equation doesn't tell us how to find $\mathbf{a}$, since the misfit $\mathbf{e}$ is unknown. We therefore need to add another requirement or condition in order to find a unique set of polynomial coefficients $\mathbf{a}$.

The key idea is to realize that the best interpolating polynomial (or the best "curve-fit") is the one that produces the smallest possible misfit. See Fig. 3.2, which shows the same 4 data points as before and one possible choice for the interpolating polynomial. Quite clearly, just by looking at it, we could definitely find better polynomials of the same order n (i.e., better choices of $\mathbf{a}$). Mathematically, we would define "better" polynomials as those which produce smaller misfits $\mathbf{e}$. Since $\mathbf{e}$ is a vector, we have to use the norm of it $\|e\|$ to measure how large or small it is. We then have

$$\mathbf{a} = \arg\min \|\mathbf{e}\| \, ,$$

which is the mathematical statement saying that the best $\mathbf{a}$ (i.e., polynomial) is the one that produces the smallest size of the misfits as measured by the norm $\|e\|$.

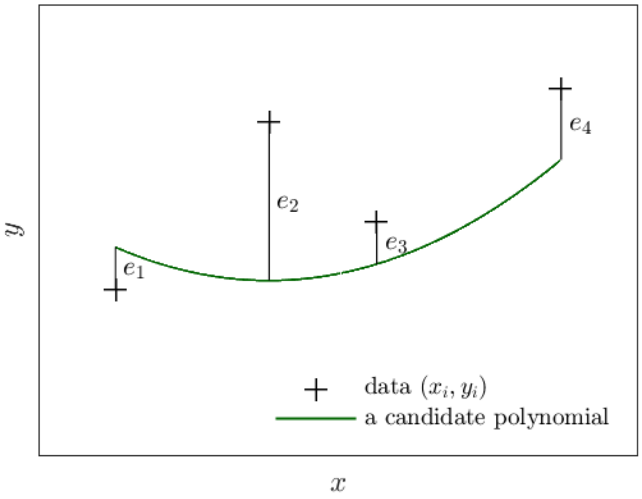

Figure 3.2: Interpolation problem with misfits e_i defined as the difference between the interpolating function and the data at each point.

At the minimum, we have the condition that

$$\frac{\partial \|\mathbf{e}\|}{\partial a_j} = 0 \ , \quad j = 0, 1, \ldots, n \, ,$$

i.e., that the quantity we are trying to minimize ($\|\mathbf{e}\|$) must have zero derivative in the direction of every variable that we are trying to find the optimal value of (the set of a_j coefficients). We can write this in vector notation as

$$\nabla_{\mathbf{a}} \|e\| = \begin{pmatrix} \partial/\partial a_0 \\ \partial/\partial a_1 \\ \vdots \\ \partial/\partial a_n \end{pmatrix} \|e\| = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \ , \tag{3.4}$$

where $\nabla_{\mathbf{a}}$ is the gradient in the space of coefficients $\mathbf{a}$.

It is useful to pause here and consider where we are. We started with an overdetermined system of N equations for $n+1$ unknowns. By defining the misfit vector $\mathbf{e}$ and requiring that it's norm should be minimized, we arrive at Eqn. (3.4) which consists of exactly $n+1$ equations. In other words, the statement of minimal total misfit produces a problem in which the number of equations and unknowns are perfectly matched, and we can solve it – regardless of what polynomial order n we choose or which norm we use.

The only remaining question is what norm to use: some choices are

$$\|e\|_1 = \sum_{i=1}^{N} |e_i| \ , \quad \|e\|_2 = \sqrt{\sum_{i=1}^{N} e_i^2} \ , \quad \|e\|_\infty = \max |e_i| \, .$$

The by-far most common choice is the second one, for which the optimal $\mathbf{a}$ can be found analytically. This approach is called the "least-squares" approach.

3.2.2 Least-squares method

The least-squares solution to the equation $A\mathbf{a} = \mathbf{y} + \mathbf{e}$ is given by $\mathbf{a} = \arg\min \|\mathbf{e}\|_2$. We then need to insert the norm $\|\mathbf{e}\|_2$ into Eqn. (3.4) to find the final equation for $\mathbf{a}$. Note that

$$\begin{aligned}\|\mathbf{e}\|_2^2 &= \mathbf{e}^T\mathbf{e}\\ &= (A\mathbf{a}-\mathbf{y})^T(A\mathbf{a}-\mathbf{y})\\ &= \mathbf{a}^TA^TA\mathbf{a} - \mathbf{a}^TA^T\mathbf{y} - \mathbf{y}^TA\mathbf{a} + \mathbf{y}^T\mathbf{y}\\ &= \mathbf{a}^TA^TA\mathbf{a} - 2\mathbf{a}^TA^T\mathbf{y} + \mathbf{y}^T\mathbf{y}\end{aligned}$$

(in the last step, note that $\mathbf{a}^TA^T\mathbf{y} = \mathbf{y}^TA\mathbf{a}$ since it's a scalar value). The gradient of this is

$$\nabla_{\mathbf{a}}\|\mathbf{e}\|_2^2 = 2A^TA\,\mathbf{a} - 2A^T\mathbf{y}\,.$$

Inserting this into Eqn. (3.4) then yields the final least-squares equation (or "normal equation")

$$A^TA\,\mathbf{a} = A^T\mathbf{y}\,.$$

(note: requiring the gradient of $\|\mathbf{e}\|$ or $\|\mathbf{e}\|^2$ to be zero is the same thing).

3.3 Splines

A spline is a collection of polynomials where each polynomial is defined in a single interval and where the polynomials are forced to match each other as well as possible at the data points. The most commonly used spline is the cubic one, in which the interpolating function in the ith interval $x \in [x_i, x_{i+1}]$ could be defined as

$$g_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + a_{i,3}x^3\ ,\quad i = 1, 2, \ldots, N-1\,.$$

For cubic splines, both the first and second derivatives are matched at the data points. An example is shown in Fig. 3.3, which clearly shows the smoothness at the data points.

With $N-1$ polynomials and 4 coefficients in each, we have $4N-4$ unknown coefficients; we then need $4N-4$ equations in order to find those coefficients. Forcing every polynomial to match the data at both ends of each interval yields the conditions

$$\begin{array}{rcl} g_i(x_i) &=& y_i \\ g_i(x_{i+1}) &=& y_{i+1}\end{array}\ ,\quad i = 1, 2, \ldots, N-1\,. \tag{3.5}$$

We need additional conditions, and impose that the first and second derivatives are continuous at the interior data points. To figure out what this means in terms of equations, the easiest way is to consider data point i, which has polynomial $g_{i-1}(x)$ to its left and polynomial $g_i(x)$ to its right. In other words,

$$\begin{array}{rcl} g'_{i-1}(x_i) &=& g'_i(x_i) \\ g''_{i-1}(x_i) &=& g''_i(x_i)\end{array}\ ,\quad i = 2, 2, \ldots, N-1\,. \tag{3.6}$$

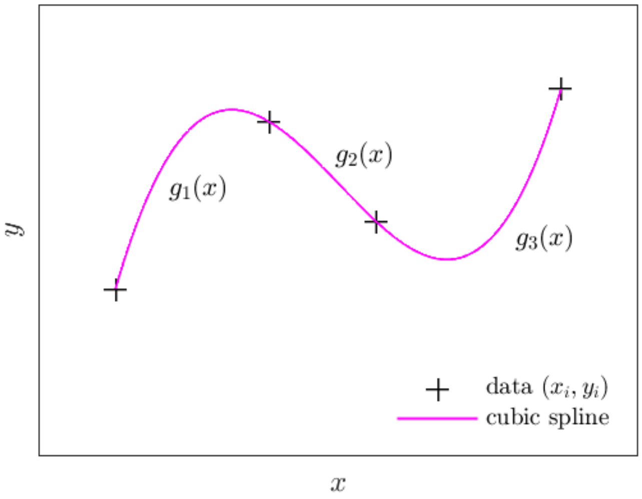

Figure 3.3: Interpolation using a cubic spline.

In total we then have $2(N-1)+2(N-2) = 4N-6$ equations, and therefore need 2 additional equations. This is most naturally done by imposing something at both ends of the spline. There are multiple options, including:

- the "clamped" spline, in which the first derivative at the end $g_1'(x_1)$ is forced to a user-provided value; this approach is natural in situations where the first derivative at the end is known from some application-specific insight, for example at adiabatic walls (the temperature gradient is zero) or in clamped beams (the slope of the beam is fixed);
- the "naturel" spline, in which the second derivative at the end $g_1''(x_1) = 0$; this mimics the way a plastic ruler is forced to bend around nails on a board (the nails being the data points), with the ruler having no bending moment and thus no curvature at the ends;
- the "not-a-knot" condition, in which $g_1''' = g_2'''$ (note: for cubic polynomials, the third derivative is constant, hence no need to specify which x_i point to use for each polynomial).

Splines are often a happy medium between the piecewise linear interpolation and the global polynomial. They are twice continuously differentiable, but retain a mostly local nature. Furthermore, they can be implemented in a way that is computationally very efficient, on par with the methods described above. For these reasons, they are frequently used in practice.

3.3.1 Derivation of cubic spline equations

The equations needed to find all spline coefficients $a_{i,j}, i = 1, \dots, N-1, j = 0, \dots, 3$ could be found from the equations and conditions provided above, which would create a system of equations of block-diagonal form. While that could certainly be solved efficiently, there is a more elegant derivation which also leads to a more elegant implementation.

The starting point is to realize that a cubic polynomial $g_i(x)$ will have a second derivative $g_i''(x)$ that varies linearly between data points. Making use of the linear interpolation formula in Eqn. (3.1), we then have

$$g_i''(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i}\, G_i + \frac{x - x_i}{x_{i+1} - x_i}\, G_{i+1}\,,$$

where G_i and G_{i+1} are the unknown parameters (or coefficients) of this line (as it happens, these parameters are actually equal to the second derivative of the polynomials at the corresponding points). For convenience we define $h_i = x_{i+1} - x_i$ as the length of each interval, and then integrate twice to get

$$g_i(x) = \frac{(x_{i+1} - x)^3}{6h_i}\, G_i + \frac{(x - x_i)^3}{6h_i}\, G_{i+1} + a_{i,1}x + a_{i,0}\,.$$

This is a cubic polynomial with built-in continuity of the second derivative at the interior data points. At this point we have $2(N-1)$ unknown coefficients $a_{i,0}$ and $a_{i,1}$, and N unknown coefficients G_i; a total of $3N-2$ unknowns. This makes sense, as we have implicitly used $N-2$ conditions to match the second derivative at the interior data points.

The $2N-2$ conditions from Eqn. (3.5) to match the data can be used to solve for the $a_{i,0}$ and $a_{i,1}$ coefficients, which yields

$$\begin{aligned} g_i(x) = {} & \left[\frac{(x_{i+1} - x)^3}{6h_i} - \frac{(x_{i+1} - x)h_i}{6}\right] G_i + \left[\frac{(x - x_i)^3}{6h_i} - \frac{(x - x_i)h_i}{6}\right] G_{i+1} \\ & + \frac{x_{i+1} - x}{h_i}\, y_i + \frac{x - x_i}{h_i}\, y_{i+1}\,. \end{aligned}$$

We now have the N G_i coefficients left, and we have used the matching of the function values and the second derivatives; the only remaining condition to use is the matching of the first derivatives at the interior $N-2$ points. The first derivative is

$$g_i'(x) = \left[-\frac{(x_{i+1} - x)^2}{2h_i} + \frac{h_i}{6}\right] G_i + \left[\frac{(x - x_i)^2}{2h_i} - \frac{h_i}{6}\right] G_{i+1} + \frac{y_{i+1} - y_i}{h_i}\,.$$

Matching the first derivative according to Eqn. (3.6) at the interior points then yields

$$\frac{h_{i-1}}{6} G_{i-1} + \frac{h_{i-1} + h_i}{3} G_i + \frac{h_i}{6} G_{i+1} = \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\ , \quad i = 2, 3, \ldots, N-1\,. \tag{3.7}$$

This is the final equation for cubic splines. Note the tri-diagonal structure, i.e., that the equation for point i involves G_{i-1}, G_i, and G_{i+1}. Solving a tri-diagonal matrix can be done very quickly and with minimal coding effort, which is the main reason for defining the cubic spline from the starting point of the linearly varying second derivative.

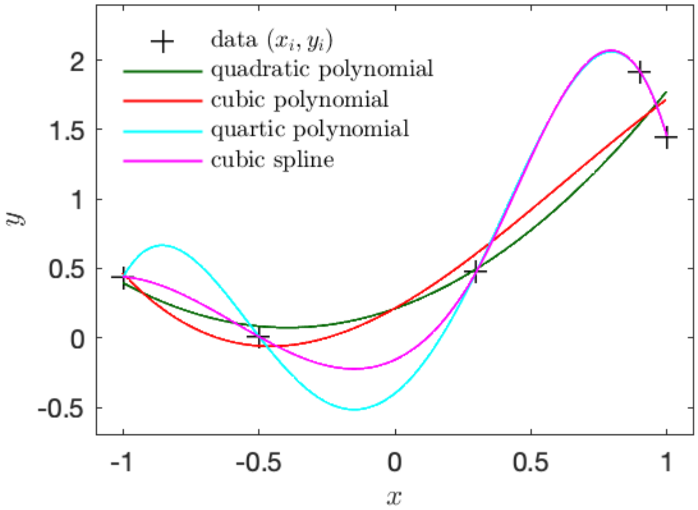

Figure 3.4: Example of different interpolation methods applied to the same data.

Equation (3.7) needs to be supplemented by two end conditions, which are encoded on the first and last row of the matrix-vector equation. The first three rows of the final matrix-vector equation are, for the special case of a "natural" end condition,

$$\begin{pmatrix} 1 & 0 & \dots & & & \\ \frac{h_1}{6} & \frac{h_1+h_2}{3} & \frac{h_2}{6} & 0 & \dots & \\ 0 & \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & 0 & \dots \\ \vdots & & & & & \end{pmatrix} \begin{pmatrix} G_1 \\ G_2 \\ G_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{y_3-y_2}{h_2} - \frac{y_2-y_1}{h_1} \\ \frac{y_4-y_3}{h_3} - \frac{y_3-y_2}{h_2} \\ \vdots \end{pmatrix} .$$

Example 3.1: interpolation of data with an outlier

Consider the given data $x = \{-1.0, -0.5, 0.3, 0.9, 1.0\}$ and $y = \{0.44, 0.01, 0.48, 1.92, 1.45\}$ which is shown along with several different interpolating functions in Fig. 3.4. The data contains an outlier (the 4th point), which makes the highest order polynomial highly oscillatory. The cubic spline is equally oscillatory around the outlier, but becomes better behaved just a few points away. The quadratic and cubic polynomials which are found through least-squares fitting avoid the oscillatory behavior.

3.4 Multi-dimensional interpolation

So far this chapter has covered only the case of one-dimensional interpolation, but let us now consider interpolation in multiple dimensions. We will cover only the case of two dimensions, as higher-dimensional cases are a straightforward extension of that. We will also only cover the case of piecewise linear interpolation.

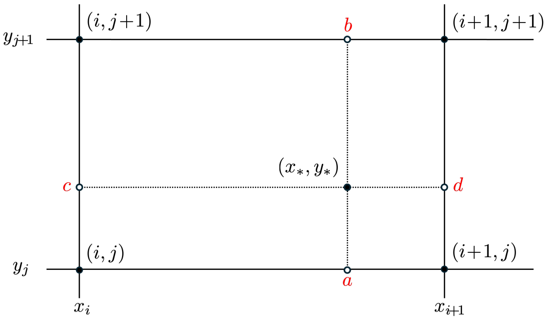

Figure 3.5: Interpolation in two dimensions on a Cartesian grid.

In two dimensions we have the known data $f(x_l, y_l)$ at a sequence of grid points (x_l, y_l) for $l = 1, 2, \ldots, N$, and we seek to find the value $f(x_*, y_*)$ at some point (x_*, y_*). In the most general case we have no idea how the data is organized; it could be a random list of points in no particular order. More commonly our data would be organized in some way. The following subsections cover a few common cases.

3.4.1 Cartesian grids

A rather common case is to have a structured Cartesian grid where x coordinates depend only on the index i, y coordinates depend only on the index j, and thus we have x_i for $i = 1, 2, \ldots, N_i$, y_j for $j = 1, 2, \ldots, N_j$, with known data $f_{i,j} = f(x_i, y_j)$.

A small part of such a grid is shown in Fig. 3.5. The first step when doing piecewise linear interpolation is to identify which element (or rectangle) our target point (x_*, y_*) resides within. For a Cartesian grid this is easy: we simply loop over i until we find $x_i \le x_* < x_{i+1}$ and then repeat this for j until we find $y_j \le y_* < y_{j+1}$. This is the stage of the process that is sketched in Fig. 3.5.

We then do one-dimensional interpolation along the grid lines using Eqn. (3.1) to find the values of f_a and f_b as

$$\begin{aligned} f_a &= \frac{x_{i+1} - x_*}{x_{i+1} - x_i} f_{i,j} + \frac{x_* - x_i}{x_{i+1} - x_i} f_{i+1,j} \,, \\ f_b &= \frac{x_{i+1} - x_*}{x_{i+1} - x_i} f_{i,j+1} + \frac{x_* - x_i}{x_{i+1} - x_i} f_{i+1,j+1} \,, \end{aligned}$$

followed by a one-dimensional interpolation in the vertical direction (we simply swap x and i for y and j) as

$$f_* = \frac{y_{j+1} - y_*}{y_{j+1} - y_j} f_a + \frac{y_* - y_j}{y_{j+1} - y_j} f_b .$$

The complete interpolation formula is more compact if we introduce the scaled coordinates

$$\xi = \frac{x_* - x_i}{x_{i+1} - x_i} \;,\; \eta = \frac{y_* - y_j}{y_{j+1} - y_j},$$

where we note that $1 - \xi = (x_{i+1} - x_*)/(x_{i+1} - x_i)$ and similarly with $1 - \eta$; we then

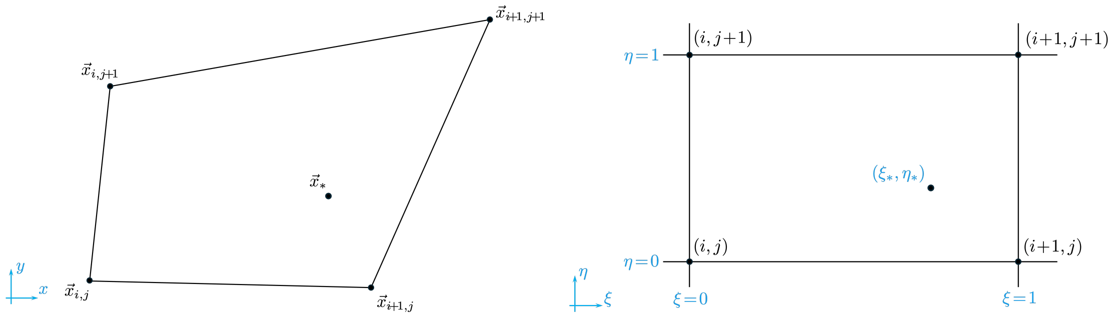

Figure 3.6: Interpolation in two dimensions on a non-Cartesian grid in physical (left) and computational (right) space.

get the complete interpolation formula as

$$\begin{aligned} f_* &= (1-\eta)f_a + \eta f_b \\ &= (1-\eta)\left[(1-\xi)f_{i,j} + \xi f_{i+1,j}\right] + \eta\left[(1-\xi)f_{i,j+1} + \xi f_{i+1,j+1}\right] \\ &= (1-\xi)(1-\eta)f_{i,j} + \xi(1-\eta)f_{i+1,j} + (1-\xi)\eta f_{i,j+1} + \xi\eta f_{i+1,j+1}. \end{aligned} \tag{3.8}$$

In this case we first interpolated in x and then in y; we could also do the opposite and first interpolate in y to find f_c and f_d and then interpolate in x to find f_*; the final interpolated value is identical (this can be verified by comparing the expressions). This successive use of one-dimensional interpolation operations is called “bilinear interpolation” (or “trilinear” in 3D).

3.4.2 Structured but non-Cartesian grids

A more complicated scenario occurs when the data is defined on a structured but non-Cartesian grid (meaning that grid points can be organized with (i,j) indexing; mathematically, the grid is topologically identical to a Cartesian grid). A single element of such a grid is sketched in Fig. 3.6.

We cannot use bilinear interpolation directly in this case, since it is ambiguous where we would place the analogoues of points “a” and “b” from Fig. 3.5. The solution is to create a “computational space” (ξ,η) in which our quadrilateral element is a Cartesian object, and then use bilinear interpolation in that space instead. This is sketched in Fig. 3.6. The element could be taken as any rectangular shape in computational space, but the math is simplified by taking the element specifically as the unit square; by doing this, the coordinates ξ and η are exactly the scaled coordinates in the bilinear interpolation formula (3.8). Therefore, we only need to find the coordinates (ξ_*,η_*) of the interpolation target, and can then use Eqn. (3.8).

To find these coordinates, we need to define the coordinate mapping between the computational and physical spaces. A bilinear mapping is

$$\vec{x}_* = \vec{x}_{i,j} + (\vec{x}_{i+1,j} - \vec{x}_{i,j})\xi + (\vec{x}_{i,j+1} - \vec{x}_{i,j})\eta + (\vec{x}_{i+1,j+1} + \vec{x}_{i,j} - \vec{x}_{i+1,j} - \vec{x}_{i,j+1})\xi\eta,$$

where one can verify that this returns the correct coordinates at the four grid points. This is actually identical to formula (3.8), but applied to the coordinates rather than the function. Our task now is to solve for ξ and η for a given $\vec{x}_*$. We have two equations (the mapping for both x and y) and hence this is doable. The equations can be combined into a quadratic equation for either ξ or η, which can be solved analytically; the equation gets rather long, though, and is therefore not shown here. Or we could use a root-finding method to find ξ and η numerically. Either way, once this is done, we apply the bilinear interpolation formula (3.8) in computational space to find the interpolated value.

We actually skipped the issue of how to identify which quadrilateral element the interpolation target point is within. This is not trivial for a non-Cartesian grid, and raises the question of what it means to be inside an element. One answer is the following: imagine that we traverse around the element in a counter-clockwise direction; if the point is always on our left, it is inside the element. Conversely, if it is to our right for even a single element edge, it is not inside the element. An similar answer is to consider the vectors from the target point to each of the grid points (or corners of the quadrilateral): if a counter-clockwise rotation takes the vector to the (i, j) node to the vector to the $(i + 1, j)$ node (and similarily for the other corners), the point is inside the element. This could be formalized as: if all of the conditions

$$\begin{aligned}
[(\vec{x}_{i,j} - \vec{x}_*) \times (\vec{x}_{i+1,j} - \vec{x}_*)] \cdot \vec{e}_z > 0, \\
[(\vec{x}_{i+1,j} - \vec{x}_*) \times (\vec{x}_{i+1,j+1} - \vec{x}_*)] \cdot \vec{e}_z > 0, \\
[(\vec{x}_{i+1,j+1} - \vec{x}_*) \times (\vec{x}_{i,j+1} - \vec{x}_*)] \cdot \vec{e}_z > 0, \\
[(\vec{x}_{i,j+1} - \vec{x}_*) \times (\vec{x}_{i,j} - \vec{x}_*)] \cdot \vec{e}_z > 0
\end{aligned}$$

are satisfied, the point is inside the element.

Chapter 4

Modal expansions and filtering

It is sometimes very useful to express a function $f(x)$ as a linear combination of basis functions $g_i(x)$, i.e., as

$$f(x) = \sum_i a_i g_i(x) \, ,$$

where a_i are the coefficients multiplying each basis function.

To see why this might be useful, consider how we express a vector as a linear combination of basis vectors, for example

$$\mathbf{f} = \sum_i a_i \mathbf{e}_i \, ,$$

where $\mathbf{e}_i$ could be (for example) unit vectors in the x-, y-, and z-directions. Having expressed the vector $\mathbf{f}$ in terms of a set of basis vectors, we can more easily gain an appreciation for what “makes up” the vector: e.g., we could see that a vector may lie mostly in the $x - y$ plane if the a_3 component is small.

For continuous functions we use basis *functions* rather than *vectors*, but apart from that there is no real difference between these cases. More to the point (in the context of this book), once we approximate continuous functions by a set of discrete values, the function becomes a finite-dimensional vector anyway.

There are many possible choices for basis functions, each of which could be used in a modal expansion. Sets of basis functions that are orthogonal make for particularly efficient and useful expansions. The most common of these is to use sines and cosines as basis functions, which is then called the Fourier transform; this is covered first in this chapter. We will then cover the concept of filtering, which is the process of changing the frequency content of a function.

4.1 Discrete Fourier transform

While many interpolation methods are based on polynomials, in some situations there are better choices available. One such scenario is if: (i) the data is uniformly spaced, and (ii) the data is assumed to be *periodic* over some distance L. In that case, a discrete Fourier series or Fourier transform can be very powerful.

A function is periodic over a distance L (or, "L-periodic") if it repeats itself after that distance regardless of where you start. Mathematically, an L-periodic function satisfies

$$f(x) = f(x \pm L)\,,\ \forall x\,.$$

If this function is known only on a uniformly spaced discrete set of points $x_j = hj, j = 1, 2, \ldots, N$, with $h = L/N$, then the condition of periodicity becomes

$$f_j = f(x_j) = f(x_j \pm L) = f_{j\pm N}\,,\ j = 1, 2, \ldots, N\,.$$

In other words, the discrete function becomes N-periodic.

Under these conditions we can write the function f_j as a Fourier series or a linear combination of "modes" as

$$f_j = \sum_{l=-N/2}^{N/2-1} \hat{f}_l\, e^{\imath k_l x_j}\,,$$

where $\hat{f}_l$ is the amplitude of each mode (the "modal amplitude" or "modal coefficient" or "Fourier coefficient"), $k_l = 2\pi l/L$ is the wavenumber or angular frequency of the mode number l, $e^{\imath k_l x_j}$ is the mode shape or basis function, and $\imath = \sqrt{-1}$. Note that the wavelength of mode l is $\lambda_l = L/l$ (i.e., there are l periods within the distance L) and that the wavenumber $k_l = 2\pi/\lambda_l$. The Nyquist wavelength, i.e., the smallest wavelength that can be resolved on the grid, is $\lambda_{\text{Nyquist}} = 2h$, for which $k_{\text{Nyquist}} = \pi/h$. The Nyquist criterion can thus be written as $|kh| \leq \pi$.

This idea of writing the function as a linear combination of modes is general; for the specific case where the basis functions are sines and cosines, we call it a Fourier series. It is important to note that the modal amplitudes $\hat{f}_l$ have the same units (or dimension) as f_j itself, and that the wavenumber k_l has units of inverse length. If f_j is a function of time rather than spatial position, one would equivalently use time t in place of location x and thus the angular frequency ω instead of the wavenumber k.

Since $x_j = hj = (L/N)j$, we see that $k_l x_j = 2\pi lj/N$. Thus the discrete Fourier transform becomes

$$f_j = \sum_{l=-N/2}^{N/2-1} \hat{f}_l\, e^{\imath 2\pi lj/N}\,, \tag{4.1}$$

in which there is no information required about the grid-spacing h. Eqn. (4.1) is generally referred to as the "backward" transform.

To derive the "forward" transform, we multiply Eqn. (4.1) by $e^{-\imath 2\pi mj/N}$ and sum over all points j; this yields (after swapping the order of the summations)

$$\sum_{j=1}^{N} e^{-\imath 2\pi mj/N} f_j = \sum_{j=1}^{N} e^{-\imath 2\pi mj/N} \sum_{l=-N/2}^{N/2-1} \hat{f}_l\, e^{\imath 2\pi lj/N} = \sum_{l=-N/2}^{N/2-1} \hat{f}_l \underbrace{\sum_{j=1}^{N} e^{\imath 2\pi (l-m)j/N}}_{g_{lm}}\,.$$

The term labelled g_{lm} is particularly important. First note that it depends on l and m (the sum over j removes dependence on j). If $l = m$, the term inside the sum is 1, and thus the whole sum equals N. If $l \neq m$, the sum is over an integer number of periods of a trigonometric function, and therefore the sum is exactly zero. In other words, we have

$$g_{lm} = \begin{cases} N\,, & l = m\,, \\ 0\,, & l \neq m\,. \end{cases}$$

This is an expression of the concept of orthogonality: just like two vectors in real space can be orthogonal to each other, in this case two basis functions can also be orthogonal to each other. The mathematical definition of orthogonality is that the inner product between them is zero: the summation over all grid points j is the inner product in this case.

Returning to our derivation, we then get that

$$\sum_{j=1}^{N} e^{-\imath 2\pi mj/N} f_j = N\hat{f}_m\,.$$

Re-arranging this yields the "forward" transform

$$\hat{f}_m = \frac{1}{N}\sum_{j=1}^{N} f_j\, e^{-\imath 2\pi mj/N}\,. \tag{4.2}$$

The combination of Eqns. (4.1) and (4.2) form a discrete Fourier transform pair; they allow for a function to be expressed either as f_j in real space or as $\hat{f}_l$ in transformed (or wavenumber, or frequency) space. It is important to realize that there is no approximation involved, both expressions are exactly valid and a Fourier transform does not involve any loss of information.

4.1.1 Energy spectrum

The orthogonality of the modes or basis functions also implies that the "energy" in the signal can be decomposed into a sum of contributions from each mode. To see this, we first define the energy of a discrete signal as

$$\text{energy} = \frac{1}{N}\sum_{j=1}^{N} |f_j|^2 = \frac{1}{N}\sum_{j=1}^{N} f_j f_j^*\,,$$

i.e., we define the energy as the mean-square of the signal. Note that energy is meant in a mathematical sense here. Inserting the backward Fourier transform (4.1) for both f_j and f_j^* yields

$$\begin{aligned}\text{energy} &= \frac{1}{N}\sum_{j=1}^{N}\left(\sum_{l=-N/2}^{N/2-1}\hat{f}_l\, e^{\imath 2\pi lj/N}\right)\left(\sum_{m=-N/2}^{N/2-1}\hat{f}_m\, e^{\imath 2\pi mj/N}\right)^* \\ &= \frac{1}{N}\sum_{j=1}^{N}\sum_{l=-N/2}^{N/2-1}\hat{f}_l\, e^{\imath 2\pi lj/N}\sum_{m=-N/2}^{N/2-1}\hat{f}_m^*\, e^{-\imath 2\pi mj/N}\,.\end{aligned}$$

Note that we used m rather than l as the summation index for the f_j^* part, to avoid confusion between the two sums. Our goal now is to reach a final expression where $E = \sum_l \ldots$, i.e., where the energy in the signal is expressed as a sum over all modes – that way, the contribution from every mode adds up to the total. To achieve this, we change the order of the sums and get

$$\text{energy} = \sum_{l=-N/2}^{N/2-1} \hat{f}_l \sum_{m=-N/2}^{N/2-1} \hat{f}_m^* \, \frac{1}{N} \underbrace{\sum_{j=1}^{N} e^{\imath 2\pi (l-m) j/N}}_{g_{lm}} \, .$$

The same g_{lm} factor appears again, and just like before it is non-zero only for $l = m$. Therefore

$$\text{energy} = \sum_{l=-N/2}^{N/2-1} \hat{f}_l \hat{f}_l^* = \sum_{l=-N/2}^{N/2-1} \left| \hat{f}_l \right|^2 ,$$

and we see that the energy of the signal is the sum of the squared magnitude of all modal coefficients. This is called Parseval's theorem, and allows us to ask questions like "how much does each frequency (or wavenumber) contribute to the total energy?". The collection of all squared modal amplitudes is called the energy spectrum.

4.1.2 Real-valued functions

If the function f_j is real-valued, then $f_j = f_j^*$ where f^* is the complex conjugate of f. In that case the Fourier coefficients must satisfy $\hat{f}_l^* = \hat{f}_{-l}$ for all l. To see this, we take the complex conjugate of $\hat{f}_l$ as

$$\hat{f}_l^* = \left(\frac{1}{N} \sum_{j=1}^{N} f_j \, e^{-\imath 2\pi l j/N} \right)^* = \frac{1}{N} \sum_{j=1}^{N} f_j^* \, e^{\imath 2\pi l j/N} = \frac{1}{N} \sum_{j=1}^{N} f_j \, e^{-\imath 2\pi (-l) j/N} = \hat{f}_{-l} \, .$$

Since this is true for all $l = -N/2, \ldots, N/2-1$, we specifically have that $\hat{f}_0 = \hat{f}_0^*$; in other words, the modal amplitude or Fourier coefficient for mode 0 must be real-valued. The same is true for mode $N/2$, since

$$\hat{f}_{N/2} = \frac{1}{N} \sum_{j=1}^{N} f_j \, e^{-\imath 2\pi (N/2) j/N} = \frac{1}{N} \sum_{j=1}^{N} f_j \, e^{-\imath \pi j} = \frac{1}{N} \sum_{j=1}^{N} f_j \, e^{\imath \pi j} = \hat{f}_{-N/2}^* \, .$$

So, for a real-valued function $f_j, j = 1, 2, \ldots, N$, the Fourier transform has real-valued $\hat{f}_0$ and $\hat{f}_{-N/2}$, with the remaining coefficients satisfying $\hat{f}_l = \hat{f}_{-l}^*, l = 1, 2, \ldots, N-1$. Note that the transformed function now has $N-1$ independent complex-valued numbers and 2 real-valued ones, meaning a total of N real-valued "pieces of data". This is the same as required to express a real-valued f_j in real space, of course.

4.2 Filtering

The concept of filtering refers to the idea of changing the frequency content of a signal. For example, think of the way you can change the amount of bass (low frequencies) or treble (high frequencies) on a music player. The prototype problem is therefore: given a discrete signal f_j described on a uniform grid $x_j = hj, j = 1, 2, \ldots, N$, we want to find a "filtered" signal $\overline{f}_j$ in which we have modified the frequency content.

The easiest and most straightforward scenario is for periodic signals using the Fourier transform. After applying the forward transform (4.2) which provides the modal coefficients $\hat{f}_m$, we can multiply the modal amplitudes with a function G_m and then apply the backward transform (4.1) to find the filtered signal $\overline{f}_j$. Modes for which $G_m < 1$ are attenuated while modes for which $G_m > 1$ are amplified. The most common situation in practice is the low-pass filter in which we want to keep the lowest frequencies (or wavenumbers) untouched while attenuating the highest frequencies: this would call for a G_m function that is 1 for the lowest modes and < 1 (or even zero) for the highest modes.

If the signal f_j is not periodic, or if we want to avoid using the Fourier transform for some other reason, we have to apply the filter operation in real space. A rather general form for doing that is

$$\overline{f}_j = \sum_{l=a}^{b} c_l f_{j+l} \,, \tag{4.3}$$

where the set of $\{c_l\}$ are coefficients that define the filter.

To analyze this filter, we make the idealized assumption that a Fourier transform applies. Note that this assumption is only made when *analyzing* the method, not when *implementing* it. Assuming that the signal f_j can be written as a backward Fourier transform (4.1) then yields, after inserting this into the filter formula (4.3),

$$\begin{aligned} \overline{f}_j &= \sum_{l=a}^{b} c_l \left(\sum_{m=-N/2}^{N/2-1} \hat{f}_m \, e^{\imath 2\pi m(j+l)/N} \right) = \sum_{m=-N/2}^{N/2-1} \hat{f}_m \sum_{l=a}^{b} c_l \, e^{\imath 2\pi m(j+l)/N} \\ &= \sum_{m=-N/2}^{N/2-1} \underbrace{\hat{f}_m \left(\sum_{l=a}^{b} c_l \, e^{\imath 2\pi m l/N} \right)}_{\hat{\overline{f}}_m} e^{\imath 2\pi m j/N} \,, \end{aligned}$$

where we can identify the quantity marked with the underbrace as the Fourier coefficient of the filtered signal $\overline{f}_j$. We can then define the "transfer function" as the ratio of each Fourier coefficient after and before the application of the filter, i.e.,

$$G_m = \frac{\hat{\overline{f}}_m}{\hat{f}_m} = \sum_{l=a}^{b} c_l \, e^{\imath 2\pi m l/N} \,.$$

Since the wavenumber of mode m is $k_m = 2\pi m/L = 2\pi m/(Nh)$, the exponent can also be written as $\imath k_m h l$. If we suppress the mode index m, we then get the transfer function as

$$G(kh) = \sum_{l=a}^{b} c_l \, e^{\imath kh\, l} \,,$$

which shows how it is a function of the product kh, the wavenumber scaled by the grid-spacing h. The maximum possible value of this scaled wavenumber is π, which is called the Nyquist criterion. We are therefore interested in how the transfer function $G(kh)$ behaves for $kh \in [0, \pi]$, or from infinitely long waves to waves with just two points per wavelength.

To see how to interpret the transfer function $G(kh)$, we can consider the simple example when f_j is a single cosine mode, i.e.,

$$f_j = \cos(kx_j) = Re\left\{ e^{\imath kx_j} \right\} .$$

By definition, the filtered version of this signal is then

$$\overline{f}_j = Re\left\{ G(kh)\, e^{\imath kx_j} \right\} = Re\left\{ |G|\, e^{\imath\theta}\, e^{\imath kx_j} \right\} = |G| \, \cos(kx_j + \theta) \,.$$

We then see that the magnitude of the transfer function $|G|$ controls how the amplitude of the modal coefficient is changed, and that the phase angle $\theta = \tan^{-1}(Im\{G\}/Re\{G\})$ controls the phase shift of the filtering process. In other words, a real-valued transfer function $G(kh)$ introduces no phase shift.

Example 4.1: design of a low-pass filter

Imagine that we want to design a low-pass filter, meaning that we want to choose coefficients c_l that, when used in Eqn. (4.3), leaves low wavenumbers (or frequencies) untouched and attenuates as much as possible the highest wavenumbers (or frequencies). We thus take the following requirements:

- $G(0) = 1$, which means that infinitely long waves are unaffected by the filter. This implies that
$$\sum_l c_l \, e^{\imath 0 l} = \sum_l c_l = 1 \,.$$
- $G(\pi) = 0$, which means that the highest wavenumber in the signal (due to the Nyquist criterion) is completely removed. This implies that
$$\sum_l c_l \, e^{\imath \pi l} = \sum_l c_l (-1)^l = 0 \,.$$
- To avoid phase shifts, we want $G(kh)$ to be real-valued for all wavenumbers. To see what this implies, we can write
$$\begin{aligned} G(kh) = \sum_l c_l \, e^{\imath khl} &= c_0 + \sum_{l>0} \left[c_l \, e^{\imath khl} + c_{-l} \, e^{-\imath khl} \right] \\ &= c_0 + \sum_{l>0} \left[\frac{c_l + c_{-l}}{2} \left(e^{\imath khl} + e^{-\imath khl} \right) + \frac{c_l - c_{-l}}{2} \left(e^{\imath khl} - e^{-\imath khl} \right) \right] \\ &= c_0 + \sum_{l>0} \left[\frac{c_l + c_{-l}}{2} \, 2\cos(khl) + \frac{c_l - c_{-l}}{2} \, 2\imath \sin(khl) \right] \,. \end{aligned}$$

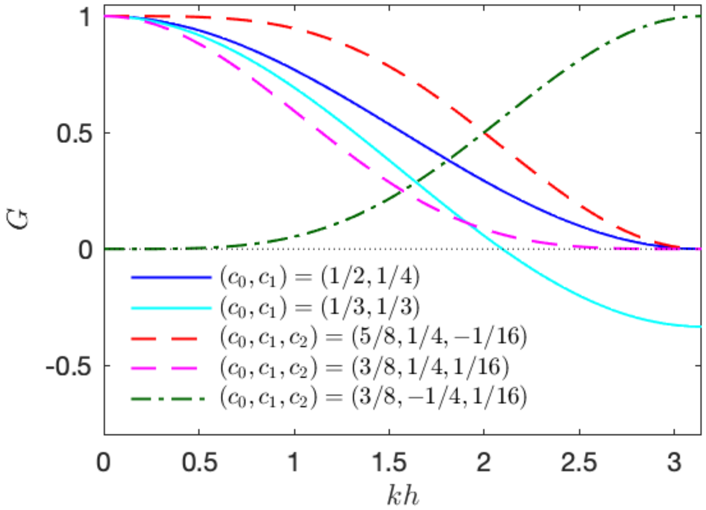

Figure 4.1: Transfer functions for some filters.

> This shows that, if $c_l = c_{-l}$, then the transfer function is purely real. In other words, a filter operation only introduces a phase shift (non-real G) if the filter is not symmetric. The most common situation where that occurs is for one-sided filters, which can occur either near boundaries or if one is applying the filter in real time with only data from the past.

Solving for the coefficients yields $(c_0, c_1) = (1/2, 1/4)$. This is therefore the "best" low-pass filter possible using only those 3 stencil points, if by "best" we mean "adheres to the requirements we stated". Perhaps surprisingly, the solution was *not* $c_0 = c_1 = 1/3$, which one might have expected on grounds of this providing "more" averaging; however, the transfer function of that filter reaches negative values for large kh, as seen in Fig. 4.1.

If one allows for a larger stencil (i.e., including more neighboring points), then one can impose additional requirements. The transfer functions of two filters using 5-point stencils are also shown in Fig. 4.1. These were designed to produce $d^2G/d(kh)^2 = 0$ at either $kh = 0$ (the red dashed line) or $kh = \pi$ (the magenta dashed line).

The filter corresponding to the green dash-dotted line in Fig. 4.1 is a high-pass filter, leaving the highest wavenumbers rather untouched while attenuating the lowest ones.

Finally, note that the low-pass filters in Fig. 4.1 have different wavenumber characteristics: e.g., the filter corresponding to the red dashed line leaves more modes with relatively large amplitude, while the filter corresponding to the magenta dashed line attenuates more modes. We would then say that the former has a larger "cut-off frequency" or "cut-off wavenumber".

Chapter 5

Integration

Consider the problem of computing an integral

$$I = \int_a^b f(x)dx\,.$$

The numerical approximation of the integral I is called "quadrature". Most quadrature methods are based on the idea of dividing the full integral into many small segments, to then approximate the integral in each segment, and to finally sum the results. This concept is sketched in Fig. 5.1. The general idea is that the approximation in each segment introduces error, but that this error will be smaller if the segments are smaller.

Following the sketch, we would divide the full interval $[a, b]$ into N segments with $x_0 = a$, $x_N = b$, and the points in between in increasing order. The jth segment would then cover $[x_{j-1}, x_j]$ and be of width $h_j = x_j - x_{j-1}$, where the sizes of the segments might vary. With this, the full integral is exactly

$$I = \sum_{j=1}^{N} I_j \tag{5.1}$$

where

$$I_j = \int_{x_{j-1}}^{x_j} f(x)dx\,.$$

There are two different scenarios in which we may want to compute an integral numerically in practice:

1. We may know the function $f(x)$ only through its values at a discrete set of data points, in which case we could not possibly find the analytical integral. In this case, the choice of how to decompose the interval $[a, b]$ into segments (how many, where to place the dividing points) is most likely dictated by where we have data.
2. We may know the function $f(x)$ analytically, but not be able to find the analytical integral if $f(x)$ is sufficiently complex. In this case, we have full freedom in how to decompose the interval $[a, b]$.

Some quadrature algorithms can handle both types of problems, but some algorithms are suitable only to the second class of problems if they are based on the ability to insert or choose data points wherever one wants.

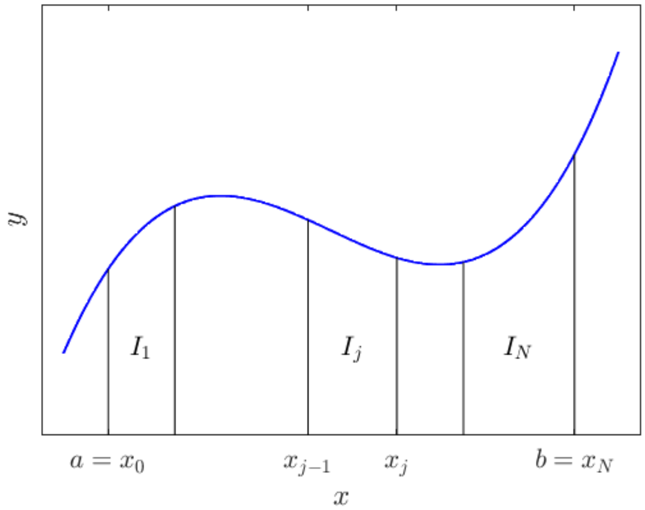

Figure 5.1: Integration problem with division into N segments.

5.1 Trapezoidal and midpoint rules

The perhaps most intuitive quadrature method is the trapezoidal rule, in which the integral over a single segment is approximated as

$$I_{j,\text{trap}} = h_j \, \frac{f(x_{j-1}) + f(x_j)}{2} \, .$$

This amounts to approximating the function by a straight line in each segment (i.e., piecewise linear interpolation between the data points at the segment boundaries), as sketched in Fig. 5.2.

Another intuitive alternative is the midpoint rule, in which the integral over a single segment is approximated as

$$I_{j,\text{midpoint}} = h_j \, f\left(\frac{x_{j-1} + x_j}{2}\right) \, .$$

To simplify notation, we often write $x_{j-1/2} = (x_{j-1} + x_j)/2$. The integral over each segment is thus approximated by a rectangle with height $f(x_{j-1/2})$, i.e., the function value at the midpoint of the segment – hence the name of the method, of course. This is also sketched in Fig. 5.2. Note that one can also think of the midpoint rule as approximating the function by a straight line that goes through the midpoint $f(x_{j-1/2})$ (as illustrated by the dash-dotted line in the figure): the area of the resulting trapezoid is identical to the area of the rectangle, regardless of the slope of this straight line.

Having described these two methods (the numerical methods part of this book), let us next turn to numerical analysis to see what accuracy we should expect. The midpoint rule is the easiest to analyze. The true function $f(x)$ can be described in the

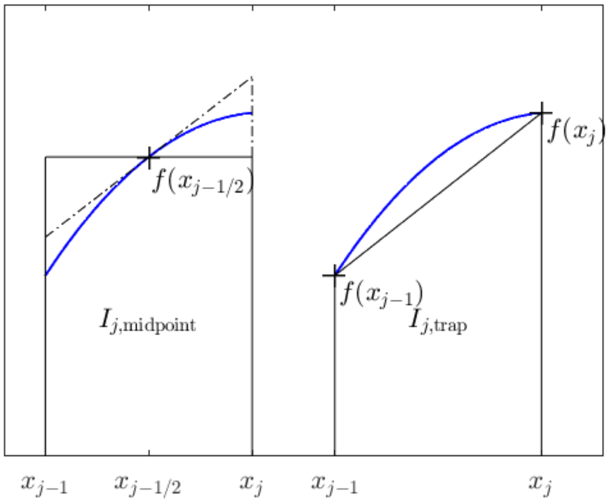

Figure 5.2: Approximate integral of a single segment using the midpoint and trapezoidal rules, with the integral approximated as the area under the black lines.

neighborhood of $x_{j-1/2}$ as

$$f(x) \approx f(x_{j-1/2}) + f'(x_{j-1/2})(x - x_{j-1/2}) + f''(x_{j-1/2})\frac{(x - x_{j-1/2})^2}{2} + \dots .$$

With this, the true segment integral is (integrate term by term)

$$\begin{aligned} I_j &\approx \int_{x_{j-1}}^{x_j} \left[f(x_{j-1/2}) + f'(x_{j-1/2})(x - x_{j-1/2}) + f''(x_{j-1/2})\frac{(x - x_{j-1/2})^2}{2} + \dots \right] dx \\ &= \left[f(x_{j-1/2})\, x + f'(x_{j-1/2})\frac{(x - x_{j-1/2})^2}{2} + f''(x_{j-1/2})\frac{(x - x_{j-1/2})^3}{6} + \dots \right]_{x_{j-1}}^{x_j} \\ &= f(x_{j-1/2})\, h_j + f''(x_{j-1/2})\frac{h_j^3}{24} + \mathcal{O}\big(h_j^5\big) \ . \end{aligned}$$

Note that the terms with odd order derivatives disappear. This can be visually seen in Fig. 5.2, where the area between the dash-dotted and solid lines must be zero since the two triangles are identical (but with different sign of the integral); the same would be true for higher-order odd derivatives.

We note that the first term on the right-hand-side is exactly the numerical formula for the midpoint rule. We therefore have

$$I_{j,\text{midpoint}} \approx I_j - f''(x_{j-1/2})\frac{h_j^3}{24} + \mathcal{O}\big(h_j^5\big) \ .$$

Summing over all segments yields

$$I_{\text{midpoint}} = \sum_{j=1}^{N} I_{j,\text{midpoint}} \approx \sum_{j=1}^{N} \left[I_j - f''(x_{j-1/2})\frac{h_j^3}{24} + \mathcal{O}\big(h_j^5\big) \right] \ .$$

The sum over the first term is the exact integral. For convenience when handling the other terms, let's define the average segment size $\overline{h} = (b-a)/N$. The remainder term is then $\mathcal{O}\left(\overline{h}^4\right)$ since the sum over all segments implies a factor of N. We then find the error in the full integral to be

$$\varepsilon_{\text{midpoint}} = |I_{\text{midpoint}} - I| \approx \left| \sum_{j=1}^{N} f''(x_{j-1/2}) \frac{h_j^3}{24} + \mathcal{O}\left(\overline{h}^4\right) \right| .$$

So the error is controlled by the second derivatives at the segment midpoints and the size of each segment. The second derivative may change sign over the interval, so it is possible that the first error term is very small (even zero). This will only happen for very special functions and choices of the segment spacing, so it's more interesting and useful to find an approximate bound on the error. This is done by taking the absolute value operation inside the sum using the triangle inequality (recall: $|x+y| \leq |x| + |y|$), after which we get

$$\varepsilon_{\text{midpoint}} \lesssim \frac{1}{24} \sum_{j=1}^{N} \left| f''(x_{j-1/2}) \right| h_j^3 + \mathcal{O}\left(\overline{h}^4\right) .$$

This is not a true bound on the error since we still have all the higher-order terms, but it becomes an approximate bound in the limit of $N \to \infty$. We still would like to simplify it further, specifically by removing the summation. We can multiply and divide by $\overline{h}^2$ to get

$$\begin{aligned} \varepsilon_{\text{midpoint}} &\lesssim \frac{1}{24} \overline{h}^2 \sum_{j=1}^{N} \left| f''(x_{j-1/2}) \right| \left(\frac{h_j}{\overline{h}} \right)^2 h_j + \mathcal{O}\left(\overline{h}^4\right) \\ &\leq \frac{1}{24} \overline{h}^2 (b-a) \left\{ \left| f'' \right| \left(\frac{h_j}{\overline{h}} \right)^2 \right\}_{\max} + \mathcal{O}\left(\overline{h}^4\right) . \end{aligned}$$

This is our final result for the error of the midpoint rule formula. The error is proportional to the full integration interval $b-a$, which makes sense. It is proportional to the square of the average segment size $\overline{h}$, implying that, if you double the number of segments, the error will decrease by a factor of 4: therefore, the midpoint rule is "second order accurate". Finally, the error depends on the curvature (=second derivative) of the function: for more highly curved functions, the error bound is larger.

The presence of the "relative" segment size $h_j/\overline{h}$ shows that one can decrease the error by using smaller segments in regions where the function has large curvature: we will use this below in section 5.3 to derive an adaptive quadrature method.

The error analysis for the trapezoidal rule is a bit more involved. The process for the analysis of the midpoint rule was that we expressed the function $f(x)$ in terms of a Taylor expansion around $x_{j-1/2}$, which then meant that the midpoint rule formula appeared directly as the first term of the Taylor expansion. This was crucial in the analysis, as we needed to find a relationship between the exact segment integral I_j and the formula of the method. To achieve the same for the trapezoidal method, we will need to have $f(x_{j-1})$ and $f(x_j)$ in the expansion, which seems impossible. We can do this by also using Taylor expansions of $f(x_{j-1})$ and $f(x_j)$. This is left as an exercise, and we simply give the final result here which is that

$$\varepsilon_{\text{trap}} = |I_{\text{trap}} - I| \lesssim \frac{1}{12}\overline{h}^2(b-a)\left\{|f''|\left(\frac{h_j}{\overline{h}}\right)^2\right\}_{\max} + \mathcal{O}\left(\overline{h}^4\right) .$$

This error bound is exactly twice larger than for the midpoint rule – this might at first seem surprising (surely approximating a function by trapezoids is better than by rectangles?), but can make graphical sense by looking at Fig. 5.2. Recall that the midpoint rule can be viewed as computing the integral of any trapezoid for which the top boundary goes through the midpoint, at any angle. The error for the midpoint rule is then the area between the dash-dotted curve and the function $f(x)$. Similarly, the error for the trapezoidal rule is the area between the solid curve and the function. The curvature of the function means that the latter area is larger, in fact exactly twice larger.

At the end of the day, the choice between the midpoint and trapezoidal rules is often made based on where one has discreetly known data. For example, if one has sampled data and wants to compute the integral between a set of data points, the trapezoidal rule is the most natural.

5.2 Simpson's rule

Simpson's rule is a natural extension of the trapezoidal rule, in which a parabola rather than a line is fitted to the data points in a segment. Since a parabola has three coefficients, we need three data points: thus Simpson's rule requires data at x_{j-1}, $x_{j-1/2}$, and x_j. Fitting a parabola and integrating yields

$$I_{j,\text{simpson}} = h_j \frac{f(x_{j-1}) + 4f(x_{j-1/2}) + f(x_j)}{6} .$$

It is important to realize that the derivation of this formula made use of the fact that $x_{j-1/2}$ is exactly halfway between x_{j-1} and x_j, and thus one could not apply Simpson's rule to two subsequent segments with unequal widths.

The error of Simpson's rule is

$$\varepsilon_{\text{simpson}} = |I_{\text{simpson}} - I| \lesssim \frac{1}{2880}\overline{h}^4(b-a)\left\{|f''''|\left(\frac{h_j}{\overline{h}}\right)^4\right\}_{\max} + \mathcal{O}\left(\overline{h}^6\right) .$$

It is 4th order accurate, meaning: if you double the number of segments, then the error will decrease by a factor of 2^4.

Note that we have defined the segment size h_j here as $x_j - x_{j-1}$. It is quite common in other parts of the literature to instead view Simpson's rule as covering two segments (that must be of equal size): in this case, the formulae above need to be adjusted accordingly.

5.3 Adaptive quadrature

The analysis of the errors in the trapezoidal and midpoint rule methods showed that the error for a single segment is proportional to the second derivative of the function and the cube of the segment size. This shows that, if we use a uniform "grid" (meaning, division of data points or segment sizes), then those segments with the largest (in magnitude) second derivative f'' will contribute the most to the error. In fact, for segments where the function is very close to a straight line, the error contribution is close to zero. This type of thinking suggests that it is not optimal to have the same segment sizes everywhere: rather, we should use smaller segments where the function varies a lot (actually, where it has large second derivative) and larger elsewhere.

This is the basis for so-called adaptive quadrature methods, in which we seek to subdivide segments until their error contribution is sufficiently small. The whole idea relies on us having the ability to insert data points where we want, and therefore adaptive quadrature is not useful in situations where one has discrete data: instead, it is useful when we are trying to integrate a "difficult" function.

The key idea in adaptive quadrature is to choose a quadrature method and then try to compute the integral for every single segment to an error less than some tolerance. Imagine that we want to compute

$$I_{\text{seg}} = \int_{x_l}^{x_r} f(x)dx$$

using the trapezoidal method. If we treat the whole interval using a single segment, the error is $\sim (x_r - x_l)^3 \, |f''|$. We could then try to estimate the second derivative in order to estimate the magnitude of the error. An alternative and arguably simpler approach is to instead re-compute the integral using a single segment of Simpson's rule, for which we know that the error is $\sim (x_r - x_l)^5 |f''''|$. Importantly, the difference between the two approximations has the error of the trapezoidal method as its leading error term, and we can therefore use this difference as an approximate estimate of the error in the trapezoidal method. Equivalently, we can say that the error in the trapezoidal method is

$$|I_{\text{seg,trap}} - I_{\text{seg,exact}}| \approx |I_{\text{seg,trap}} - I_{\text{seg,simp}}| \, ,$$

i.e., we can view Simpson's rule as an approximation of the exact answer in order to estimate the error. If the difference between the low- and high-order methods is larger than the tolerance, the segment is split into two and the process is repeated for each of those two segments. Clearly the same idea could be used with any combination of methods, provided that one of them is of higher-order accuracy than the other.

Algorithm 5.1 gives the essential structure of the implementation. Note that this uses so-called "recursive" programming, in which the function calls itself – this is key to a compact implementation. Also note that we assign a lower tolerance to the computation of each sub-segment: if the errors of the two segments add up (i.e., if they don't fortuitously cancel), this means that the overall tolerance for the segment is still within the tolerance.

A natural question is whether it would be better to use the Simpson's rule result rather than the trapezoidal one when assigning the value of the segment integral –

Algorithm 5.1: adaptive quadrature

```
**function** $I$ = adaptiveQuadrature($f(x), x_l, x_r$, tol)
$I_{\text{trap}} = (x_r - x_l)(f(x_l) + f(x_r))/2$
$x_m = (x_l + x_r)/2$
$I_{\text{simp}} = (x_r - x_l)(f(x_l) + 4f(x_m) + f(x_r))/6$
if $|I_{\text{trap}} - I_{\text{simp}}| <$ tol,
    $I = I_{\text{trap}}$
else,
    $I_1$ = adaptiveQuadrature($f(x), x_l, x_m$, tol/2)
    $I_2$ = adaptiveQuadrature($f(x), x_m, x_r$, tol/2)
    $I = I_1 + I_2$
end
return $I$
```

surely it would be better to use the more accurate approximation? While there is some point to this, the real beauty of the adaptive quadrature is that it only stops when the difference between the high- and low-order methods is sufficiently small, and thus it doesn't really matter which one we take.

Chapter 6

Differentiation

Imagine that we have known data at a discrete set of points and want to compute the derivative at a specific point. In other words, we know $f_j = f(x_j)$ at a set of $x_j, j = 1, \ldots, N$, organized in increasing order, and we want to compute an approximate value of $f'_j = df/dx|_{x_j}$. We have two broad ways to approach this problem:

1. We can fit an interpolating function to the data, and then take the analytical derivative of this interpolating function. We could use any of the methods discussed in chapter 3. If the data is periodic, we could also use the Fourier transform discussed in chapter 4 as an interpolating function.
2. We can compute an approximate derivative without ever explicitly fitting an interpolating function based on the definition of a derivative as the limit of $(f(x+h) - f(x))/h$ or $(f(x+h) - f(x-h))/(2h)$ or similar as $h \to 0$; we would then use formulae like this but with a finitely large step size h. This type of approach is called "finite differencing".

6.1 Introduction to finite difference schemes

Let us start with using intuition to derive some finite difference schemes. Imagine that we want to compute the first derivative at x_j, i.e., $f'(x_j) = f'_j$. The following schemes (note: "scheme" is synonymous with "numerical formula") are then quite intuitive:

$$\begin{aligned} f'_{j,\text{left}} &= \frac{f_j - f_{j-1}}{x_j - x_{j-1}}\,, \\ f'_{j,\text{right}} &= \frac{f_{j+1} - f_j}{x_{j+1} - x_j}\,, \\ f'_{j,\text{cent}} &= \frac{f_{j+1} - f_{j-1}}{x_{j+1} - x_{j-1}}\,. \end{aligned} \tag{6.1}$$

The first method is biased to the left, the second is its mirror-image that is biased to the right. The last method is symmetric around x_j, using data from both sides. An example illustrating these different approximations of the derivative is shown in Fig. 6.1.

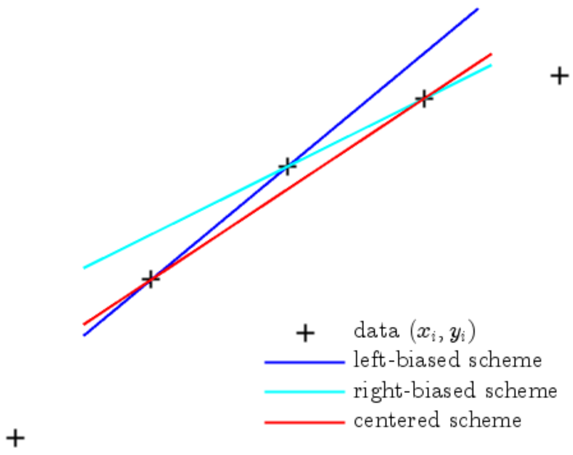

Figure 6.1: Example comparing the derivatives computed using the three different formulae in Eqn. (6.1).

Now, all of these three schemes will compute valid approximations of the true first derivative $f'(x_j)$, so which should we use in practice? It depends on the context and what accuracy we desire. For example, the left-biased scheme could be useful in a situation where we are sampling a signal in time and want to compute the derivative on-the-fly: in that case, f_{j+1} is unknown, and hence only the left-biased scheme would be useful. Similarly, both the left- and right-biased schemes could be useful when computing derivatives near the edges of where we have data.

Having said all this, assuming there are no restrictions on which scheme we can use, which of these three schemes is the most accurate? This is where numerical analysis comes in, to say something about the expected behavior of errors for these three methods.

Let us analyze what the errors are for the three examples given in Eqn. (6.1). To simplify the analysis, we assume that the data points are uniformly spaced, i.e., that $x_{j+1} - x_j = x_j - x_{j-1} = h$. Taylor expansions of both f_{j-1} and f_{j+1} around x_j can be written as

$$f_{j\pm1} = f_j \pm f_j' h + f_j'' \frac{h^2}{2} \pm f_j''' \frac{h^3}{6} + f_j'''' \frac{h^4}{24} + \dots .$$

Inserting these into the formula for $f'_{j,\text{cent}}$ then yields

$$\begin{aligned}
f'_{j,\text{cent}} &= \frac{f_{j+1} - f_{j-1}}{2h} \\
&\approx \frac{\left(f_j + f'_j h + f''_j \frac{h^2}{2} + f'''_j \frac{h^3}{6} + f''''_j \frac{h^4}{24}\right) - \left(f_j - f'_j h + f''_j \frac{h^2}{2} - f'''_j \frac{h^3}{6} + f''''_j \frac{h^4}{24}\right)}{2h} \\
&= \frac{2f'_j h + 2f'''_j \frac{h^3}{6} + \mathcal{O}\big(h^5\big)}{2h} \\
&= f'_j + \frac{h^2}{6} f'''_j + \mathcal{O}\big(h^4\big) \ .
\end{aligned}$$

This shows that the numerical scheme (or "formula" or "recipe") $f'_{j,\text{cent}}$ for estimating the derivative is equal to the exact derivative $f'_j = f'(x_j)$ plus an infinite sequence of error terms composed of higher and higher derivatives and powers of h. As the grid-spacing $h \to 0$, the values of all the higher derivatives remain the same; therefore, for h smaller than some "critical" value (which is probably very small), only the first error term $(h^2/6)f'''_j$ remains and all other terms are vanishingly small. When that happens, we are in the so-called "asymptotic range of convergence": by definition, the error is described by only a single term. Having only a single error term is important since it prevents cancellation effects between different terms. In that asympotic range, we can therefore say that, if the grid-spacing h is divided by 2, then the error in the approximation of the first derivative will be reduced by a factor of 4 due to the h^2 factor. We therefore say that the $f'_{j,\text{cent}}$ scheme is "second-order accurate".

If we repeat the same exercise for the left- and right-biased schemes in Eqn. (6.1), we find that

$$\begin{aligned}
f'_{j,\text{left}} &\approx f'_j - \frac{h}{2} f''_j + \mathcal{O}\big(h^2\big) \ , \\
f'_{j,\text{right}} &\approx f'_j + \frac{h}{2} f''_j + \mathcal{O}\big(h^2\big) \ .
\end{aligned}$$

So these methods also estimate the exact derivative f'_j in the limit of $h \to 0$, but for finite values of h they have different errors. Specifically, they are "first-order accurate": when reducing h by half, the error is reduced by half as well.

Having performed this analysis, we return to the original question of which of these three schemes is the most accurate? We actually can't say, in general. What we *can* say is that the error for the central scheme is reduced faster as $h \to 0$. So there exists some critical grid-spacing where, for all h smaller than that critical grid-spacing, the central scheme will be more accurate than the left- or right-biased schemes.

6.2 General finite difference schemes on uniform grids

The previous section described a work flow in which we intuitively defined some schemes and then analyzed their accuracy. In the present section, we will approach the problem more systematically, by defining a general scheme and then find the coefficients through numerical analysis. In addition, we will consider derivatives of arbitrary order, i.e., not just the first derivative. To keep things simple, we will assume a uniform grid-spacing $h = x_j - x_{j-1}$ for all j throughout this section.

Imagine that we want to compute an approximation of $d^n f/dx^n$ at x_j, i.e., the nth derivative of f, using a formula of type

$$\left.\frac{d^n f}{dx^n}\right|_{x_j,\mathrm{num}} = \frac{1}{h^n}\sum_{l=a}^{b} c_l f_{j+l}\,. \tag{6.2}$$

Before continuing, let us think about this formula. The set of $\{c_l\}$ are the coefficients of the scheme, at this point unknown. In order to be of general use, these coefficients must not have any dimension; therefore, there *must* be a factor of h^{-n} in the formula, since h is the only thing with the same dimension as x here. The numbers a and b define which data points are included in the scheme. For example, $a = -1$ and $b = 2$ would imply that four data points are included, with one to the left and two to the right. The collection of data points included in the scheme is often referred to as the "stencil".

We now have two scenarios: either (i) we have been given a set of coefficients $\{c_l\}$ and want to analyze what accuracy they imply; or (ii) we have to find values for the coefficients in order to derive a formula for a particular derivative with a particular accuracy. We actually proceed in the same way for both scenarios, since the strategy for finding coefficient values is to perform the accuracy analysis and then at the end ask what coefficient values would be required.

The analysis process starts by expressing all function values f_{j+l} in terms of Taylor expansions around x_j as

$$f_{j\pm l} = f_j \pm lhf'_j + \frac{l^2h^2}{2}f''_j \pm \frac{l^3h^3}{6}f'''_j + \frac{l^4h^4}{24}f''''_j + \dots\,.$$

We then insert this into the general formula (6.2) and collect terms to get

$$\begin{aligned}
\left.\frac{d^n f}{dx^n}\right|_{x_j,\mathrm{num}} &= \frac{1}{h^n}\sum_{l=a}^{b} c_l\left(f_j \pm lhf'_j + \frac{l^2h^2}{2}f''_j \pm \frac{l^3h^3}{6}f'''_j + \frac{l^4h^4}{24}f''''_j + \dots\right)\\
&= \frac{\sum_l c_l}{h^n}f_j + \frac{\sum_l lc_l}{h^{n-1}}f'_j + \frac{\sum_l l^2c_l}{2h^{n-2}}f''_j + \frac{\sum_l l^3c_l}{6h^{n-3}}f'''_j + \frac{\sum_l l^4c_l}{24h^{n-4}}f''''_j + \dots \qquad (6.3)\\
&= \sum_{m=0}^{\infty}\frac{1}{m!}h^{m-n}\left(\sum_{l=a}^{b} l^m c_l\right)\left.\frac{d^m f}{dx^m}\right|_{x_j}\,.
\end{aligned}$$

We now have a relation between the formula (left hand side) and the function and all its derivatives at x_j. If we know the values of the coefficients c_l, we simply compute each sum and then have an expression to analyze. If we instead want to find the coefficient values, we need to think about what we want the right-hand-side to be.

We first note the following: the formula will only compute an approximation to the nth derivative as $h \to 0$ under the following conditions:

- All terms with $m < n$ (i.e., terms with lower derivatives of f than the desired nth derivative) must be exactly zero; if not, they will go to infinity as $h \to 0$ due to the h^{m-n} factor. So we need

$$\sum_{l=a}^{b} l^m c_l = 0\,, \;\; m = 0, 1, \ldots, n-1\,.$$

- The term with the nth derivative must be exactly $d^n f/dx^n|_j$, meaning that we need

  $$\frac{1}{n!}\sum_{l=a}^{b} l^n c_l = 1\,.$$

  Note that the h factor disappears for this term, by construction: we argued this on dimensional grounds above, but we can now see that it is also necessary on grounds that the coefficients c_l must not depend on h.

The conditions above provide $n+1$ equations, and they must be enforced exactly. This implies that we need a minimum of $n+1$ non-zero coefficients c_l in order to have a formula for the nth derivative; equivalently, we need a stencil that includes at least $n+1$ data points. For example, it would be impossible to approximate the 4th derivative from only 3 data points.

In addition to the necessary requirements above, we can impose additional desirable attributes of our approximate formula. The obvious desirable attribute is that the error should be as small as possible. The simplest way to enforce this is to make some of the remaining error terms exactly zero, which will then enforce a certain order of accuracy. For example, if the term with the $(n+1)$th derivative is enforced to be zero, then the leading error term becomes proportional to h^2: a second-order accurate method. Or we could also enforce that the term with the $(n+2)$th derivative is zero, which leads to a third-order accurate method. In both cases, we add an additional equation which then means that we will need additional non-zero coefficients c_l: the more requirements, the larger the stencil we need.

Once we have specified the $n+1$ required and k desired conditions, we have $n+1+k$ equations. We must then choose a stencil of exactly $n+1+k$ non-zero coefficients, i.e., we have to choose the value of a or b (and then find the other such that exactly $n+1+k$ data points are involved). This then allows us to write a linear system of equations for the unknown coefficient values. An example of this process is shown below.

Example 6.1: how to solve for finite difference coefficients

Imagine that we want to find the coefficients c_l for a finite difference scheme that approximates the second derivative at x_j to the highest possible order of accuracy using a five-point completely centered stencil (i.e., using points $j-2, j-1, j, j+1, j+2$). We then have the hard (non-negotiable) requirements

$$\sum_l c_l = 0\,, \;\; \sum_l l c_l = 0\,, \;\; \frac{1}{2}\sum_l l^2 c_l = 1\,.$$

As this only provides 3 equations for the 5 unknowns, we are free to enforce 2 additional constraints. To maximize the order of accuracy, these should be

$$\sum_l l^3 c_l = 0\,,\ \sum_l l^4 c_l = 0\,.$$

The linear system of equations is then (note that the columns correspond to $l = -2, -1, 0, 1, 2$)

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \\ 16 & 1 & 0 & 1 & 16 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} c_{-2} \\ c_{-1} \\ c_0 \\ c_1 \\ c_2 \end{pmatrix}}_{\mathbf{c}} = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{pmatrix}\,,$$

which has the solution $\mathbf{c} = (-1/12, 4/3, -5/2, 4/3, -1/12)^T$.

Note that we moved the factor of 2 to the right-hand-side for convenience: when doing that, the matrix A has rows $\mathbf{l}^m$ for $m = 0, 1, \ldots$, where the vector $\mathbf{l}$ contains the values of l in the stencil.

To find out which order of accuracy this scheme has, we need to evaluate the sums in Eqn. (6.3) and see which are zero. For this case, the first non-zero error term occurs for $m = 6$ as

$$-\frac{8}{6!} h^4 \left.\frac{d^6 f}{dx^6}\right|_{x_j}\,.$$

We thus see that this scheme is 4th order accurate.

Example 6.2: how to solve for finite difference coefficients with a biased stencil

Imagine that we want to find the coefficients c_l for a finite difference scheme that approximates the first derivative at x_j to the highest possible order of accuracy using a five-point stencil that is fully biased towards the right (i.e., using points $j, j{+}1, j{+}2, j{+}3, j{+}4$). We then have the hard (non-negotiable) requirements

$$\sum_l c_l = 0\,,\ \sum_l l c_l = 0,$$

and the additional constraints (to reach maximum order of accuracy)

$$\sum_l l^2 c_l = 0\,,\ \sum_l l^3 c_l = 0\,,\ \sum_l l^4 c_l = 0\,.$$

To build the linear system of equations, we need to remember that our stencil covers $l = 0, 1, 2, 3, 4$; this determines which columns we have in the matrix. We get

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 9 & 16 \\ 0 & 1 & 8 & 27 & 64 \\ 0 & 1 & 16 & 81 & 256 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}}_{\mathbf{c}} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}\,,$$

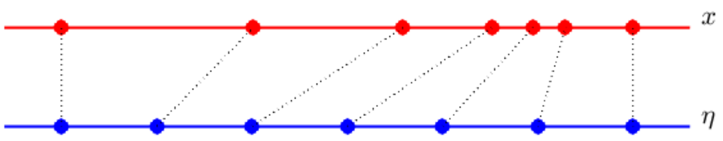

Figure 6.2: Sketch of the grid points in physical (x) and computational (η) coordinates.

which has the solution $\mathbf{c} = (-25/12, 4, -3, 4/3, -1/4)^T$, where we need to remember that the elements of $\mathbf{c}$ are $c_0, c_1, \ldots, c_4$.

6.3 Finite differences on non-uniform grids

All of the material above has assumed that the grid-spacing is the same everywhere, but in practice it is very common to use non-uniform grids. For example, if taking measurements or solving the equations for a boundary layer in fluid mechanics, most of the variation occurs close to the wall and thus it makes sense to place more measurement or solution points close to the wall than farther out.

Our derivation of the scheme coefficients above assumed a uniform grid, and the analysis is actually invalid for non-uniform grids. In principle we could repeat the analysis for a non-uniform grid, but the problem is that this would have to be done for every possible non-uniform grid, and the resulting scheme coefficients would be unique to each possible non-uniform grid. This is clearly not useful. Fortunately there is a different approach which avoids this problem. This approach is based on coordinate transformations, specifically on the idea of transforming the actual grid in physical space to a uniform grid in an imagined "computational" space. We can then apply our existing finite difference schemes in computational space since the grid there is uniform, and we can finally transform the result (the approximate derivative) back to physical space.

Consider a physical space coordinate x and a non-uniform grid $\{x_j\}$, and a computational space coordinate η and a uniform grid $\{\eta_j\}$. Examples of such a pair of grids is shown in Fig. 6.2. Now imagine that we can describe the mappings between these coordinate systems by $\eta = g(x)$ and $x = G(\eta)$. It turns out that we actually don't need to know what these functions are; we only need them as intellectual tools in the derivation. The function in real space is $f(x)$ and the same function in computational space is $F(\eta)$. Since they describe the same function, we have $f(x) = F(\eta)$ for $x = G(\eta)$ and $\eta = g(x)$.

We can then use the chain-rule of differentiation to get

$$\underbrace{\frac{df}{dx}}_{f'(x)} = \underbrace{\frac{dF}{d\eta}}_{F'(\eta)} \frac{d\eta}{dx} \, .$$

Since $F'(\eta)$ is defined in computational space where the grid-spacing is uniform, we can use any existing finite difference method to approximate it, for example,

$$F'(\eta_j) = \frac{F(\eta_{j+1}) - F(\eta_{j-1})}{2\Delta\eta} = \frac{f(x_{j+1}) - f(x_{j-1})}{2\Delta\eta} \, ,$$

where $\Delta\eta$ is the grid-spacing in computational space and where we used the fact that $f(x) = F(\eta)$ for corresponding points.

The second factor in the chain-rule $d\eta/dx$ cannot be computed directly since it is defined in physical space. However, we can differentiate $x = G(\eta) = G(g(x))$ with respect to x and get

$$\underbrace{\frac{dx}{dx}}_{1} = \underbrace{\frac{dG}{d\eta}}_{G'(\eta)} \underbrace{\frac{d\eta}{dx}}_{g'(x)} \, ,$$

which then implies

$$\frac{d\eta}{dx} = \frac{1}{G'(\eta)} \, .$$

Since $G'(\eta)$ is defined in computational space, we can use any finite difference scheme to compute it.

Inserting this into the chain-rule then yields the final expression

$$\frac{df}{dx} = \frac{F'(\eta)}{G'(\eta)} \, .$$

In other words, to compute the first derivative of a function in physical space, we use a finite difference scheme to compute the first derivative of the function in computational space, use a finite difference scheme again to compute the first derivative of the coordinate mapping, and finally divide the two results with each other.

For example, the second-order accurate central scheme is

$$\left.\frac{df}{dx}\right|_{j,\mathrm{2nd-cent}} = \frac{\frac{F_{j+1}-F_{j-1}}{2\Delta\eta}}{\frac{G_{j+1}-G_{j-1}}{2\Delta\eta}} = \frac{f_{j+1} - f_{j-1}}{x_{j+1} - x_{j-1}} \, .$$

This is very intuitive, arguably exactly what one would have guessed. However, since all the formulae here apply to *any* finite difference schemes, we can instead do it for the fourth-order accurate central scheme to get

$$\left.\frac{df}{dx}\right|_{j,\mathrm{4th-cent}} = \frac{-\frac{1}{12}f_{j+2} + \frac{2}{3}f_{j+1} - \frac{2}{3}f_{j-1} + \frac{1}{12}f_{j-2}}{-\frac{1}{12}x_{j+2} + \frac{2}{3}x_{j+1} - \frac{2}{3}x_{j-1} + \frac{1}{12}x_{j-2}} \, .$$

This is not something that one would come up with without the formality of the coordinate mapping.

As a side-point, these formulae show that the concept of the grid-spacing is actually non-unique for non-uniform grids. In the coordinate mapping approach, the "grid-spacing" is really the derivative of the coordinate x with respect to the computational space η; while the derivative of the imagined continuous function is of course unique, the fact that we can approximate it using any finite difference scheme means that the discrete value is non-unique.

The equivalent formula for the second derivative is a bit more involved. We have

$$\begin{aligned}\frac{d^2 f}{dx^2} = \frac{d}{dx}\left(\frac{df}{dx}\right) = \frac{d}{dx}\left(\frac{F'(\eta)}{G'(\eta)}\right) = \frac{d\eta}{dx}\,\frac{d}{d\eta}\left(\frac{F'(\eta)}{G'(\eta)}\right) \\ = \frac{1}{G'(\eta)}\left(\frac{F''(\eta)}{G'(\eta)} - \frac{F'(\eta)G''(\eta)}{(G'(\eta))^2}\right) \\ = \frac{F''(\eta)}{(G'(\eta))^2} - \frac{F'(\eta)G''(\eta)}{(G'(\eta))^3}\,.\end{aligned}$$

We then need four different applications of any finite difference scheme to compute approximations to F' and F'' (the first and second derivatives of the function) and G' and G'' (the first and second derivatives of the coordinate mapping). For example, using the second-order accurate central schemes for both the first and second derivatives yields

$$\left.\frac{d^2 f}{dx^2}\right|_{j,\mathrm{2nd-cent}} = \frac{f_{j+1} - 2f_j + f_{j-1}}{\left(\frac{x_{j+1}-x_{j-1}}{2}\right)^2} - \frac{f_{j+1} - f_{j-1}}{2}\,\frac{x_{j+1} - 2x_j + x_{j-1}}{\left(\frac{x_{j+1}-x_{j-1}}{2}\right)^3}\,.$$

If the grid is uniform, the coordinate mapping is linear and has zero second derivative; in that case, the second term is zero. We can therefore think of the formula as being composed of a main term (the first one) and a correction for the effect of the non-uniform grid (the second term).

6.4 Implementation as a matrix-vector product

All of the finite difference schemes discussed above have been described on a point-wise basis, with the formula given to compute the derivative at some grid point j in terms of nearby function values. Closer inspection of these formulae show that they are all linear combinations of function values, which then means that they can be written in matrix-vector form. This can sometimes be very useful in practice, when implementing these methods in actual code.

We define the vector of grid points as $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ and the vector of function values as $\mathbf{f} = (f_1, f_2, \ldots, f_N)^T$. We can then write the general formula (6.2) for the case of a uniform grid as

$$\left.\frac{d^n \mathbf{f}}{dx^n}\right|_{\mathrm{num}} = \frac{1}{h^n} D_n \mathbf{f}\,,$$

where D_n is a matrix containing the coefficients. Some examples of D_n are shown in the example below.

If the grid is non-uniform, we instead have, for the first derivative,

$$\left.\frac{d\mathbf{f}}{dx}\right|_{\text{num}} = \text{diag}\big(\mathbf{x}'\big)^{-1}\, D_1\mathbf{f}\,,$$

where

$$\mathbf{x}' = D_1\mathbf{x}\,.$$

This shows how the finite difference matrix D_1 is re-used for the computation of both derivatives. This is even more true when computing the second derivative. In practice, one can get the same benefit by implementing the differencing operation in a function.

Example 6.3: finite difference scheme in matrix form

Imagine that we want to implement a first derivative scheme in matrix form, and that we want to use the 4th-order accurate central scheme in most of the domain. This scheme has a five-point stencil using the two nearest neighbors, and thus it can be applied from the 3rd grid point forward but not to the first two grid points; we therefore get the differencing matrix

$$D_1 = \begin{pmatrix} ? & ? & ? & \dots & & & \\ ? & ? & ? & \dots & & & \\ 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 & 0 \\ 0 & 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}.$$

To complete the definition, we need to choose schemes for the first two grid points. The simplest choice is to use a second-order accurate central scheme in the second grid point, and a right-biased first-order accurate scheme in the first. This would yield

$$D_1 = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1/2 & 0 & 1/2 & 0 & 0 & 0 & 0 \\ 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 & 0 \\ 0 & 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}.$$

This would work, but would reduce the order of accuracy near the boundary. If one wants to avoid that, one can use higher-order methods for those first two grid points. For example,

$$D_1 = \begin{pmatrix} -25/12 & 4 & -3 & 4/3 & -1/4 & 0 & 0 \\ -1/4 & -5/6 & 3/2 & -1/2 & 1/12 & 0 & 0 \\ 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 & 0 \\ 0 & 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix},$$

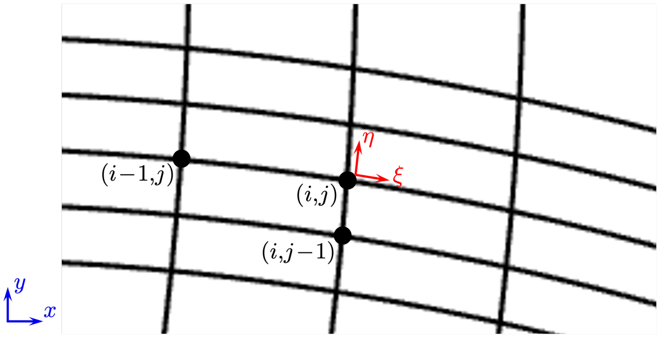

Figure 6.3: A sample two-dimensional grid with the physical and computational coordinate systems, and the indices of three grid points.

which is fourth-order accurate everywhere. Note that we had to design fourth-order accurate *biased* schemes for the first two grid points, using stencils $l = 0, 1, 2, 3, 4$ (first grid point) and $l = -1, 0, 1, 2, 3$ (second grid point); the scheme for the first grid point was actually derived in Example 6.2.

A potential problem is that this includes more than 2 neighbors in the first two grid points. In some implementations it is important to use the same number of neighbors everywhere; in that case, we could do

$$D_1 = \begin{pmatrix} -3/2 & 2 & -1/2 & 0 & 0 & 0 & 0 \\ -1/3 & -1/2 & 1 & -1/6 & 0 & 0 & 0 \\ 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 & 0 \\ 0 & 1/12 & -2/3 & 0 & 2/3 & -1/12 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}.$$

There are of course many other possible choices.

6.5 Finite differences in higher dimensions

Imagine that we have a problem defined in two dimensions with grid points $\{(x_{i,j}, y_{i,j})\}$ with indices $i = 1, \dots, N_i$ and $j = 1, \dots, N_j$. An example of such a grid is shown in Fig. 6.3. The grid points are aligned in a "mesh" such that the index i increases in one direction while the other index j increases in the other; this is called a "structured" grid. The question is how to compute the gradient $\nabla f = (\partial f/\partial x, \partial f/\partial y)^T$ at grid point (i, j)? We first notice that we cannot associate the x- and y-directions with the i- and j-indices, as the grid is not aligned with the Cartesian directions. The solution is to use a coordinate transformation from the "physical" (x, y) space to a "computational" (ξ, η) space. The idea is exactly the same as what we used for a non-uniform grid in section 6.3, but expanded to two dimensions.

The function is physical space is $f(x, y)$, while in computational space it is $F(\xi, \eta)$. These describe the same function, so $f(x, y) = F(\xi, \eta)$ whenever the (x, y) and (ξ, η)

coordinates correspond to the same point after the unknown coordinate transformation. The chain-rule of differentiation yields

$$\begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial F}{\partial \xi}\dfrac{\partial \xi}{\partial x} + \dfrac{\partial F}{\partial \eta}\dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial F}{\partial \xi}\dfrac{\partial \xi}{\partial y} + \dfrac{\partial F}{\partial \eta}\dfrac{\partial \eta}{\partial y} \end{pmatrix} = \underbrace{\begin{pmatrix} \dfrac{\partial \xi}{\partial x} & \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial \xi}{\partial y} & \dfrac{\partial \eta}{\partial y} \end{pmatrix}}_{A} \begin{pmatrix} \dfrac{\partial F}{\partial \xi} \\ \dfrac{\partial F}{\partial \eta} \end{pmatrix} . \tag{6.4}$$

The final vector is the gradient of F in computational space, which we can compute using finite differences along the ξ and η directions since the grid is perfectly Cartesian and uniform in computational space (compare this with section 6.3: there we used only the uniformity of the grid in computational space, now we also use the Cartesian nature). The matrix A contains the coordinate transformation, which can not be easily computed. However, let us write the inverse operation

$$\begin{pmatrix} \dfrac{\partial F}{\partial \xi} \\ \dfrac{\partial F}{\partial \eta} \end{pmatrix} = \underbrace{\begin{pmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{pmatrix}}_{B} \begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{pmatrix}$$

and then combine these expressions to

$$\begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{pmatrix} = AB \begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{pmatrix} ,$$

which shows that $A = B^{-1}$. The key point here is that we *can* compute B using finite differences in computational space: just like in section 6.3, we can apply our finite difference method to the coordinates x and y just as well as to the function. Taking the inverse of B then yields

$$A = \begin{pmatrix} \dfrac{\partial \xi}{\partial x} & \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial \xi}{\partial y} & \dfrac{\partial \eta}{\partial y} \end{pmatrix} = \frac{1}{\det(B)} \begin{pmatrix} \dfrac{\partial y}{\partial \eta} & -\dfrac{\partial y}{\partial \xi} \\ -\dfrac{\partial x}{\partial \eta} & \dfrac{\partial x}{\partial \xi} \end{pmatrix} . \tag{6.5}$$

We are now ready to compute the gradient. We apply our chosen finite difference scheme along both of the directions of the grid as

$$\left.\frac{\partial g}{\partial \xi}\right|_{i,j} = \frac{1}{\Delta \xi} \sum_l c_l g_{i+l,j} \,,$$
$$\left.\frac{\partial g}{\partial \eta}\right|_{i,j} = \frac{1}{\Delta \eta} \sum_l c_l g_{i,j+l} \,,$$

where g is f, x, and y; i.e., we compute 6 different derivatives in computational space. We then compute the coordinate mapping A using Eqn. (6.5), and finally compute the gradient in physical spacing using Eqn. (6.4).

The extension to three (or higher) dimensions is rather trivial, the only difficulty being to find the inverse of the coordinate mapping matrix.

Chapter 7

Analyzing random data

Data from any real experiment will be contaminated by random noise, making the measured data random in nature. Engineering decisions or scientific analysis are almost based on the underlying deterministic (or systematic) behavior, and one must therefore use averaging in order to reduce the random fluctuations in the data. In many cases it is then also important to estimate how large the random effects are.

7.1 Correlated data and auto-correlation

The covariance between two random variables X and Y is defined as

$$\mathrm{Cov}(X, Y) = E\left[(X - \mu_X)(Y - \mu_Y)\right]$$

and measures the degree of correlation between the two random variables. If the value of X is completely independent of the value of Y, then the covariance will be zero. If high values of X are more likely to occur when Y takes on high values, the covariance will be positive.

Imagine a time-dependent signal $x(t)$ or its discreet equivalent x_i. We can then ask how correlated the signal is with itself at a later time, i.e.,

$$R(\tau) = \mathrm{Cov}(x(t), x(t + \tau))$$

which we can approximate for discreet data as

$$R_k = \frac{1}{n-k} \sum_{i=1}^{n-k} (x_i - \overline{x})(x_{i+k} - \overline{x})\,.$$

This "auto-correlation" (the correlation of the signal with itself) R_k depends only on the separation between the data points, or on the time delay τ in the continuous example. If the random process that produced the data "loses memory" of its prior states after some time delay τ, we should expect the auto-correlation R to become zero for times larger than τ. In that way, the auto-correlation says something about the correlation length of a data set.

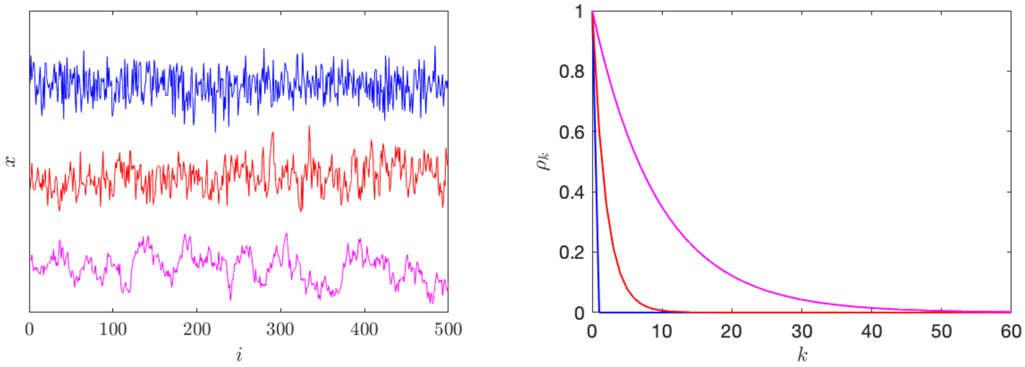

Figure 7.1: Three random signals (left) with different levels of correlation as evidenced by their auto-correlation coefficients (right).

In practice, one frequently defines a normalized auto-correlation coefficient as

$$\rho_k = \frac{\mathrm{Cov}(x_i, x_{i+k})}{\sigma_x^2} = \frac{R_k}{R_0}\,.$$

Example 7.1: correlated data

One simple way to generate a correlated random signal is as to take $x_0 = \varepsilon_0$ and then $x_i = Cx_{i-1} + \sqrt{1-C^2}\,\varepsilon_i$, $i = 2, 3, \ldots$, where $C \in [0, 1]$ is a parameter and ε_i is a random number with zero mean and unity variance. One can show that this creates a random signal x_i with auto-correlation coefficient $\rho_k = C^k$.

Three sample signals with $C = 0$ (uncorrelated), $C = 0.6$, and $C = 0.9$ are shown in Fig. 7.1 along with their auto-correlation coefficients. The signals "lose memory" of their past either immediately (for $C = 0$) or after approximately 10 and 50 time units (the other cases).

7.2 Confidence intervals for correlated data

The concept of a confidence interval is very important in the area of statistical inference, providing a probabilistic estimate of a true underlying quantity in terms of the sampled data. The most common example is the confidence interval for the expected value μ_X of a random variable X in terms of the computed sample mean $\overline{x}$ and the computed sample standard deviation s_x from n samples, as described by Eqn. (1.2) in section 1.3.4. The difficult part in computing a confidence interval is the estimation of the standard error, i.e., the standard deviation of the sample mean $\sigma_{\overline{x}}$. For uncorrelated (or statistically independent) data this is simply $\sigma_{\overline{x}} = \sigma_x/\sqrt{n}$, a formula which is taught in most introductory courses on statistics. However, the data produced in many engineering and science applications is instead highly correlated, meaning that successive data points are not independent of each other. The subject of this section is to show how one can compute the estimated standard error for correlated data.

Assume we have sampled data $x_i, i = 1, 2, \ldots, n$, which is correlated. The *standard error* is defined as the standard deviation of the sample mean, i.e., $\sigma_{\overline{x}}$. The squared standard error is

$$\sigma_{\overline{x}}^2 = E\left[(\overline{x} - \mu_{\overline{x}})^2\right] .$$

The expected value of the sample mean is the same as the expected value itself (i.e., the sample mean is unbiased), and thus $\mu_{\overline{x}} = \mu_X$. We can then insert the sample mean formula and perform manipulations as

$$\begin{aligned}
\sigma_{\overline{x}}^2 = E\left[\left(\frac{1}{n}\sum_{i=1}^{n} x_i - \mu_x\right)^2\right] &= \frac{1}{n^2}E\left[\left(\sum_{i=1}^{n}(x_i - \mu_x)\right)^2\right] \\
&= \frac{1}{n^2}E\left[\left(\sum_{i=1}^{n}(x_i - \mu_x)\right)\left(\sum_{j=1}^{n}(x_j - \mu_x)\right)\right] \\
&= \frac{1}{n^2}\sum_{i=1}^{n} E\left[(x_i - \mu_x)^2\right] + \frac{2}{n^2}\sum_{i \neq j} E\left[(x_i - \mu_x)(x_j - \mu_x)\right] .
\end{aligned}$$

We can the identify the first sum as the variance and the second sum as the covariance between different data points, which leads us to the final formula for the sample mean standard error

$$\sigma_{\overline{x}}^2 = \frac{\sigma_x^2}{n} + \frac{2}{n^2}\sum_{i \neq j} \mathrm{Cov}(x_i, x_j) .$$

The second term becomes zero for uncorrelated data since each data point is then independent of all other data points. For correlated data, the second term is most likely positive since nearby data points tend to have positive correlation (or covariance) while data points far apart tend to have zero correlation (or covariance). This then implies that the standard error for correlated data is higher than for uncorrelated data (assuming the same underlying variance σ_x^2 and number of samples n). In other words, if we use the formula for independent data, we will underpredict the true standard error, and thus our confidence interval will be smaller than it should be.

7.2.1 The batch method

The arguably simplest method for computing the standard error for correlated data is to divide the data into consequtive batches, compute the sample mean within each batch, and then treat these sample means as independent random variables. The following describes the simplest possible method; a better method was developed by Russo & Luchini (2017).

Consider splitting the data into B batches, each containing $m = n/B$ data points. The data has to be contiguous, i.e., batch b should contain the data points $i = (b -$

$1)m + 1 \to bm$. Step 1 is then to compute the sample mean for each batch, i.e.,

$$\overline{x}_b = \frac{1}{m} \sum_{i=(b-1)m+1}^{bm} x_i \ , \quad b = 1, 2, \ldots, B \, .$$

We note that the sample mean of these batch means is then

$$\overline{\overline{x}} = \frac{1}{B} \sum_{b=1}^{B} \overline{x}_b = \frac{1}{n} \sum_{i=1}^{n} x_i \, ,$$

i.e., the sample mean of the full signal is now denoted by $\overline{\overline{x}}$.

By definition, the sample variance of the batch means is

$$s_{\overline{x}}^2 = \frac{1}{B-1} \sum_{b=1}^{B} \left(\overline{x}_b - \overline{\overline{x}} \right)^2 \, .$$

Now, if the batches are sufficiently large (i.e., if m is sufficiently large), then the different batch means should be independent of each other. In that case we can approximate the variance of the sample mean of the complete data as

$$\sigma_{\overline{\overline{x}}}^2 \approx \frac{\sigma_{\overline{x}}^2}{B} \, ,$$

which implies that our formula for the estimated standard error is

$$s_{\overline{\overline{x}}} = \sqrt{\frac{1}{B(B-1)} \sum_{b=1}^{B} \left(\overline{x}_b - \overline{\overline{x}} \right)^2} \, .$$

The final question is how one should choose m, the size of each batch? The key consideration is that we need the batch means to be independent, which implies that each batch must be large enough such that most data points in a batch b are uncorrelated with the data points in the nearby batches $b-1$ and $b+1$. In other words, the batches need to be sufficiently larger than the correlation length of the signal.

7.3 Energy spectrum of a long non-periodic signals

Section 4.1.1 described how we can use the Fourier transform of a periodic signal f_j to obtain its energy spectrum, which describes how much each mode (frequency or wavenumber) contributes to the total energy. For random signals, the energy spectrum is defined as the expected value of the energy in each mode; i.e., the energy spectrum is deterministic. We thus define the energy in a signal as

$$\text{energy} = E\left[\frac{1}{N} \sum_{j=1}^{N} |f_j|^2 \right] = E\left[\sum_{l=-N/2}^{N/2-1} |\hat{f}_l|^2 \right] = \sum_{l=-N/2}^{N/2-1} E\left[|\hat{f}_l|^2 \right] \, ,$$

using the notation from section 4.1. The deterministic energy spectrum is the collection of expected squared modal amplitudes. In practice, we should therefore do the following to compute the energy spectrum: run multiple experiments, collect the signal f_j for each, apply the Fourier transform and compute the squared modal amplitudes, and finally average these squared modal amplitudes over multiple experiments.

A common situation in practice is that we have only a single instance of the signal f_j (i.e., only a single "experiment") but that this single instance is very long. We can then divide the signal into several shorter segments or batches, exactly as was done in section 7.2.1 when estimating the standard error of the sample mean. The idea is then to treat each segment as an independent experiment, and thus to Fourier transform and compute the squared modal amplitudes for each segment, and finally to average the squared modal amplitudes over all segments of the signal. The lowest frequency captured in a Fourier transform is the inverse of the length of the signal, so the process of dividing the signal into several shorter segments therefore amounts to trading some low-frequency resolution for a smoother energy spectrum. As a result, the optimal number of segments, and thus the length of each segment, is therefore problem-specific.

The Fourier transform is defined for periodic data. In situations where the data is not periodic (e.g., if obtained by measuring a random process in time), this introduces so-called aliasing errors in the Fourier transform, where energy associated with the lack of periodicity is attributed to different Fourier modes. One way to reduce these aliasing errors is to multiply the signal by a window function that tapers smoothly to zero at the ends of the signal before applying the Fourier transform. If the signal is divided into several segments, this "windowing" is applied to each segment prior to taking the Fourier transform.

The complete process for computing the energy spectrum of a long non-periodic signal is therefore as listed in Algorithm 7.1.

Algorithm 7.1: energy spectrum of a long non-periodic signal

```
divide the signal into $B$ segments
for each segment,
  multiply by a window function that tapers to zero at the ends
  compute the Fourier transform
  compute the squared modal amplitudes
end for
average the squared model amplitudes over all segments
```

7.3.1 Premultiplied spectrum

Energy spectra are usually plotted with a log-scale for the frequency (or wavenumber, or mode number). The key to interpreting an energy spectrum is Parseval's relation, i.e., the fact that the sum (or integral) of all squared modal amplitudes equals the total energy of the signal. In other words, if $F(k)$ is the energy spectral density at

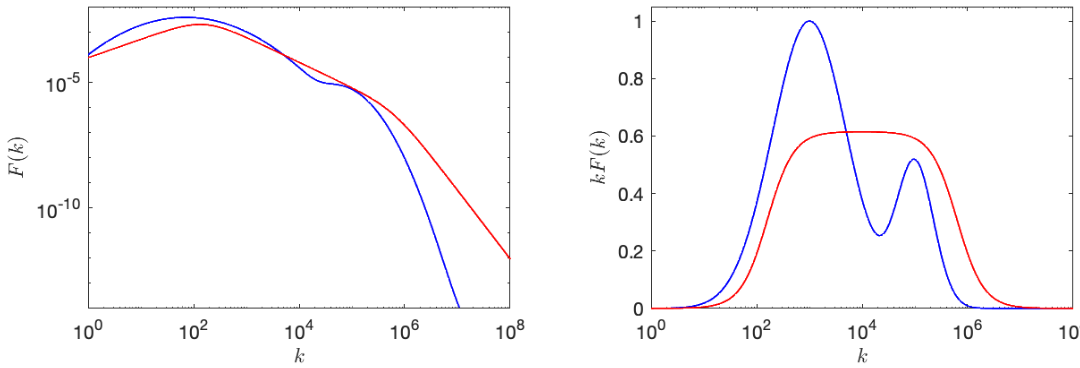

Figure 7.2: Two sample energy spectral densities $F(k)$ plotted in two different ways. Note that both spectra have the same total energy (variance).

wavenumber k, then

$$\text{energy} = \int_0^\infty F(k)dk$$

(note that $F(k)$ is the squared modal amplitude at k divided by the wavenumber "step size" Δk). If the energy spectral density $F(k)$ (or "energy spectrum") is plotted in a linear-linear plot, the area under the graph is the energy of the signal, which simplifies interpretation. In most cases, however, it makes sense to plot $F(k)$ with a logarithmic scale for the frequency or wavenumber k, in which case the area under the curve is no longer related to the energy of the signal. One can then instead plot the "pre-multiplied spectrum" $k\,F(k)$ since $dk = kd(\log k)$, and thus

$$\text{energy} = \int_0^\infty F(k)dk = \int_{-\infty}^\infty kF(k)d(\log k)\,.$$

Therefore, if $kF(k)$ is plotted with a linear axis vs k plotted with a logarithmic axis, the area under the graph is again the energy of the signal. This is illustrated in Fig. 7.2, which shows two different energy spectra plotted in two different ways. The "pre-multiplied" plot shows clearly how the blue signal has two spectral peaks while the red signal has a flat region; especially the former feature is hard to make out in the other plot.

Part II

Solving differential equations

Chapter 8

Ordinary differential equations

Differential equations occur in almost every topic area within science and engineering. Some examples of ordinary differential equations (ODEs) are

$$\frac{du}{ds} = f(u, s)$$

and

$$\frac{d^2u}{ds^2} = f(u, du/ds, s)\,,$$

where we seek the solution $u(s)$ for some interval $s \in [s_1, s_2]$. In general there will be infinitely many solutions to ODEs, and we need additional conditions to define a unique solution to the problem. In most cases these will take the form of boundary conditions, i.e., conditions on the solution u at either s_1 or s_2. For initial value problems (IVPs), the flow of information is from s_1 to s_2, and thus all boundary conditions are provided at s_1; these boundary conditions are then referred to as "initial conditions". For boundary value problems (BVPs), the flow of information goes in both directions, and thus boundary conditions are needed at both s_1 and s_2. To help build an intuitive feel for these problems, we will replace s by t (time) for IVPs and by x (space) for BVPs.

The solution u can be either scalar- or vector-valued, meaning that $u(s)$ can be either a number or a vector. The more common scenario in practice is for vector-valued solutions, and we will therefore use the notation $\mathbf{u}$ to remind ourselves that the solution is most likely a vector.

8.1 Initial value problems

The prototypical initial value problem (IVP) for the unknown $\mathbf{u}(t)$ over the interval $t \in [0, T]$ is

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}\left(\mathbf{u}, t\right)\ ,\ \ \mathbf{u}(0) = \mathbf{u}_0\,, \tag{8.1}$$

where the second part is the initial condition. The spirit of all ODE-IVP solvers is to start at the initial condition at time $t = 0$ and then march the solution forward in time by finite time steps of size h until the end time $t = T$. We thus define the solution at step n as $\mathbf{u}_n = \mathbf{u}(t_n)$, with the time step counter $n = 0, 1, \ldots$. All ODE-IVP solvers are then defined in the sense of going from step n to step $n+1$, i.e., assuming that one knows the solution $\mathbf{u}_n$, the algorithm should find the solution $\mathbf{u}_{n+1}$ at the next time level.

We note that higher-order ODEs can always be written in the general form of Eqn. (8.1) by introducing auxiliary variables that describe the lower-order derivatives. For example, the second-order ODE

$$\frac{d^2\mathbf{u}}{dt^2} = \mathbf{f}(\mathbf{u}, \mathbf{u}', t)\,,$$

where $\mathbf{u}' = d\mathbf{u}/dt$, can be written as

$$\frac{d\mathbf{v}}{dt} = \begin{pmatrix} \mathbf{f}(\mathbf{v}_2, \mathbf{v}_1, t) \\ \mathbf{v}_1 \end{pmatrix}$$

if ones defines the vector $\mathbf{v}$ as

$$\mathbf{v} = \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{u}' \\ \mathbf{u} \end{pmatrix}\,.$$

In other words, if $\mathbf{u}$ is n-dimensional, then the original n-dimensional second-order ODE can be transformed into a $2n$-dimensional first-order ODE.

8.1.1 Euler and Crank-Nicolson methods

The simplest ODE-IVP solver is the explicit Euler method (also called the forward Euler method), which is defined as

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{h} = \mathbf{f}(\mathbf{u}_n, t_n)\,, \tag{8.2}$$

where $\mathbf{u}_n$ is the current (and known) solution and $h = t_{n+1} - t_n$ is the time step. An alternative method is the implicit (or backward) Euler method, defined as

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{h} = \mathbf{f}(\mathbf{u}_{n+1}, t_{n+1})\,. \tag{8.3}$$

A third rather intuitive method is the Crank-Nicolson method which is simply the average of the explicit and implicit Euler methods, i.e.,

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{h} = \frac{1}{2}\mathbf{f}(\mathbf{u}_n, t_n) + \frac{1}{2}\mathbf{f}(\mathbf{u}_{n+1}, t_{n+1})\,. \tag{8.4}$$

The two Euler methods look very similar, differing only in whether the “right-hand-side” $\mathbf{f}$ is computed at the current or future time level. While a seemingly minor difference, this actually completely changes the nature of the method: for explicit Euler, Eqn. (8.2) provides a formula to compute the new solution $\mathbf{u}_{n+1}$; for implicit Euler, however, Eqn. (8.3) is a nonlinear system of equations that might be difficult and computationally expensive to solve. The Crank-Nicolson method also requires the solution of a nonlinear system of equations, and is therefore also categorized as an "implicit" method; similarly, all methods that do not require the solution of a nonlinear system are labeled "explicit" methods.

In general, explicit methods are simple to implement (just evaluate the formula) and computationally very cheap (again, just evaluate the formula). Their downside is that they may be unstable, defined in this context as producing solutions that go to infinity as t increases – they "blow up". In contrast, implicit methods tend to be more stable. This will be analyzed next.

8.1.2 Stability analysis – general background

In part I of this book we frequently analyzed the accuracy of different methods for differentiation, integration, etc. When solving differential equations, the question of accuracy is only half the problem – the other problem is the issue of stability, i.e., whether the method will produce a reasonable answer or whether it will produce solutions that become progressively larger in magnitude and, in normal vernacular, "blow up". We must therefore first analyse the stability attributes of any ODE-IVP method before worrying about the accuracy.

In a general sense (beyond the topics of this book), a system is (linearly) stable if a small perturbation does not grow in magnitude; conversely, it is unstable if a small perturbation does grow in magnitude. Imagine, for example, a pendulum which has two possible steady state solutions: with the pendulum straight down or straight up. Both are steady state solutions, but only the former is a stable solution: if you nudge the pendulum, it will begin oscillating back and forth, but the magnitude of this motion will not grow without bound in time.

In the context of ODE-IVP solvers, we then define a "base" solution $\mathbf{u}_{\rm base}(t)$ that solves the ODE and then study how a solution $\mathbf{u}(t) = \mathbf{u}_{\rm base}(t) + \mathbf{u}_{\rm p}(t)$ behaves. Inserting this solution into the exact ODE (8.1) then yields

$$\frac{d\mathbf{u}_{\rm base}}{dt} + \frac{d\mathbf{u}_{\rm p}}{dt} = \mathbf{f}(\mathbf{u}_{\rm base} + \mathbf{u}_{\rm p}, t) \approx \mathbf{f}(\mathbf{u}_{\rm base}, t) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right|_{\mathbf{u}_{\rm base}, t} \mathbf{u}_{\rm p}(t) + \dots ,$$

where we used a Taylor expansion to approximate the right-hand-side for small perturbations $\mathbf{u}_{\rm p}(t)$. By assuming that the base solution $\mathbf{u}_{\rm base}(t)$ solves the ODE, i.e., that $d\mathbf{u}_{\rm base}/dt = \mathbf{f}(\mathbf{u}_{\rm base}, t)$, we then get

$$\frac{d\mathbf{u}_{\rm p}}{dt} \approx \underbrace{\left.\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right|_{\mathbf{u}_{\rm base}, t}}_{A} \mathbf{u}_{\rm p}(t) \, . \tag{8.5}$$

This is an approximate ODE that describes how a small perturbation to a solution behaves, and therefore defines the stability characteristics of the original ODE. Before continuning, we make a few observations. First, we note that $A = \partial \mathbf{f}/\partial \mathbf{u}$ is a matrix of size $n \times n$, where n is the dimension of $\mathbf{u}$. This matrix is called the Jacobian matrix. Secondly, we note that this is a linear problem, since the Jacobian matrix A is defined based on the base solution rather than the perturbed solution. We can therefore use the tools of linear algebra and linear ODEs to analyze the stability of a nonlinear ODE-IVP. Finally, we note that the Jacobian matrix A is something we could, at least in principle, derive and compute from the underlying ODE problem and the assumed base solution. So, while *implementation* of an ODE-IVP solver in a code does not require the computation of A (there are exceptions to this, not covered in this book), the *stability analysis* of the ODE-IVP solver requires us to at least imagine that we know A.

Assuming that A is known, it is not trivial to look at Eqn. (8.5) and say whether the solution $\mathbf{u}_\mathrm{p}(t)$ will grow in time or not. The key analytical step then is to write the solution $\mathbf{u}_\mathrm{p}(t)$ in terms of the eigenvectors of A.

Assume that A has an eigendecomposition

$$AX = X\Lambda\,,$$

where X is a matrix that contains the eigenvectors as columns and Λ is a diagonal matrix that contains the eigenvalues along its diagonal. Assuming that the eigenvectors form a basis for the space, we can then write the solution $\mathbf{u}_\mathrm{p}(t)$ as

$$\mathbf{u}_\mathrm{p}(t) = X\mathbf{c}(t) = \sum_{i=1}^{n} \mathbf{x}_i\, c_i(t)\,.$$

The second step shows that this amounts to writing the solution $\mathbf{u}_\mathrm{p}(t)$ as a linear combination of all the eigenvectors $\mathbf{x}_i$ of A, with the coefficients being $c_i(t)$. So we have not modified the solution in any way, we are just choosing to express it in a different coordinate system – specifically, we are describing $\mathbf{u}_\mathrm{p}(t)$ in the coordinate system defined by the eigenvectors rather than the base coordinate system we used originally.

If we insert this into the stability problem (8.5) we get

$$\frac{dX\mathbf{c}}{dt} = A\,X\mathbf{c}\,.$$

Multiplying by X^{-1} and recognizing that the eigenvectors X are not dependent on time then yields

$$\frac{d\mathbf{c}}{dt} = \Lambda\mathbf{c}\,, \tag{8.6}$$

after making use of the fact that $X^{-1}AX = \Lambda$. This is the same ODE as we started with (the stability problem) but now expressed in a different coordinate system – and in this new coordinate system the different components are decoupled from each other! To see this, we can write the equation for the jth component as

$$\frac{dc_j}{dt} = \lambda_j c_j\,.$$

Now that we have used the linear algebra tool of an eigendecomposition, let us use our analytical understanding of ODE-IVPs to say that this equation has the solution

$$c_j(t) = c_j(0)e^{\lambda_j t}\,.$$

We can now say something about the stability of this problem: if even a single eigenvalue λ_j has a positive real part, then the solution will blow up. Put in a different way:

the stability problem as defined in our alternative coordinate system (i.e., Eqn. (8.6)) becomes unstable if any eigenvalue of A has a positive real part, and since the two stability problems are really one and the same this must be true for the problem in the form of Eqn. (8.5) too. Or put in a last way: the two different definitions of the solution are linked by $\mathbf{u}_{\mathrm{p}}(t) = X\mathbf{c}(t)$, and thus any statement about stability for one implies the same for the other.

The upshot of all of this linear algebra is the following: to study stability, we should study the model problem

$$\frac{dc}{dt} = \lambda c\,, \tag{8.7}$$

where λ is a scalar and $c(t)$ is scalar-valued. Also, since we know that this really represents each eigenmode in the full system of equations, we should let λ be a complex number since we know that the eigenvalues of real-valued matrices can still be (and often are) complex-valued.

The analysis then proceeds by applying a specific ODE-IVP solver to the model problem (8.7) and asking whether the solution grows in time or not. To do this, we define the amplification factor

$$G = \frac{c_{n+1}}{c_n}\,.$$

This amplification factor G is complex-valued. If $|G| > 1$, the solution will grow exponentially in magnitude and the method is unstable. If $|G| = 1$, the solution will maintain the same magnitude for ever; it might oscillate, but it will neither grow nor decay. In that case the method is marginally stable. Finally, if $|G| < 1$, then the solution will decay in time; it might oscillate, but it will eventually reach zero. In that case the method is stable.

8.1.3 Stability analysis applied to some common methods

Let us now analyse the stability of the methods discussed in section 8.1.1. For explicit Euler, application of the method defined by Eqn. (8.2) to the stability model problem (8.5) yields

$$\frac{c_{n+1} - c_n}{h} = \lambda c_n\,,$$

which we can re-arrange to find the amplification factor

$$G_{\text{exp. Euler}} = \frac{c_{n+1}}{c_n} = 1 + \lambda h\,.$$

For implicit Euler we get

$$\frac{c_{n+1} - c_n}{h} = \lambda c_{n+1}$$

which implies

$$G_{\text{imp. Euler}} = \frac{c_{n+1}}{c_n} = \frac{1}{1 - \lambda h}\,.$$

For Crank-Nicolson we get

$$\frac{c_{n+1} - c_n}{h} = \frac{1}{2}\lambda c_n + \frac{1}{2}\lambda c_{n+1}$$

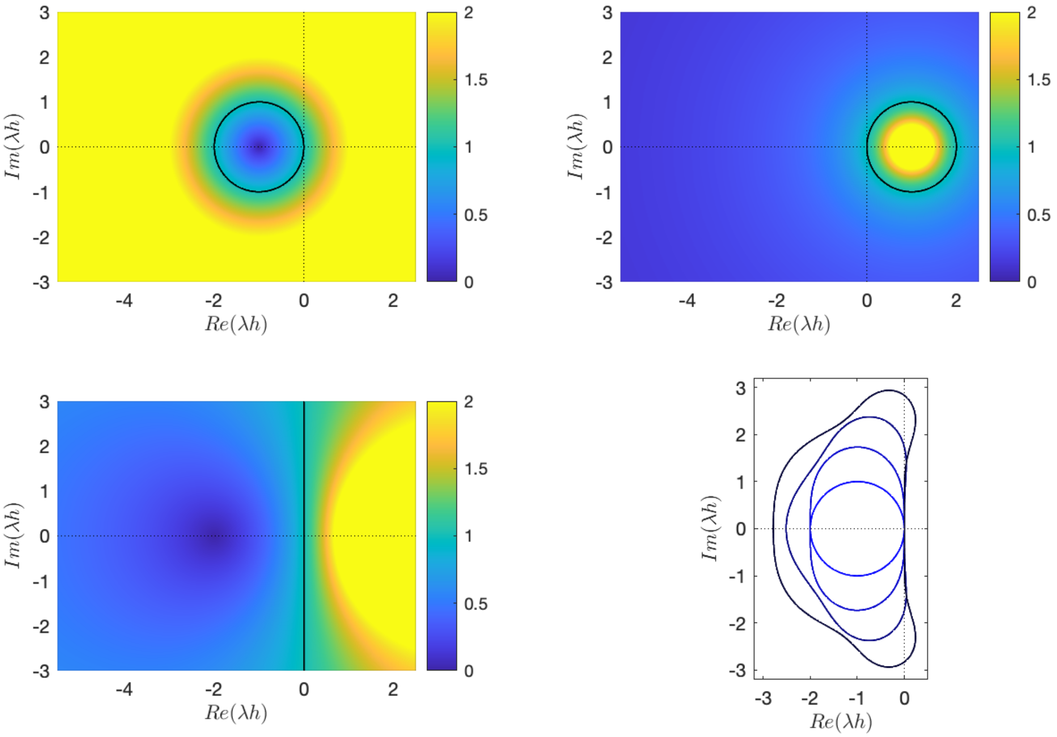

Figure 8.1: Stability plots for explicit Euler (top left), implicit Euler (top right) and Crank-Nicolson (bottom left), all showing the magnitude of the amplification factor $|G|$ in colors and the stability boundary as a solid line. Also showing the stability boundary for Runge-Kutta methods of increasing accuracy (bottom right), from RK1 (also explicit Euler) to RK4 in darker shades with larger stability regions.

which implies

$$G_{\text{Crank-Nic.}} = \frac{c_{n+1}}{c_n} = \frac{1+\lambda h/2}{1-\lambda h/2}\,.$$

The magnitude of the different amplification factors is visualized in Fig. 8.1. Stability requires $|G| \leq 1$, which is true in different regions of the complex plane for these different methods (blue colors in the figure). The implicit Euler and Crank-Nicolson methods are both stable for all λh with real parts ≤ 0. The implicit Euler method is actually stable for some λh with positive real parts too, but this is arguably more of academic interest since $Re(\lambda) > 0$ implies that the actual problem should be unstable too; arguably the notion of numerical stability only makes sense if the problem is analytically stable.

The explicit Euler method is stable only for λh values in a small circle. The term for this is that it is "conditionally stable", meaning that it is stable for some λh but not all. In contrast, the implicit Euler and Crank-Nicolson methods are "unconditionally stable", meaning that they are stable for all λh in the left half-plane (i.e., with real part ≤ 0).

We also note that the explicit Euler method is unstable for all λh that lie on the imaginary axis, i.e., with exactly zero real part. This is powerful information: it tells us that, for problems that create Jacobian matrices with purely imaginary eigenvalues, the explicit Euler method is useless!

Finally, we observe that the Jacobian eigenvalue λ and the time step h always appear as a product, meaning that their individual values are unimportant. Let's think about what this means. The stability model problem (8.7) shows that the units of λ must be the inverse of the units of h – since h is a time step (measured in seconds, say), that means that λ must be a rate (measured in 1/seconds). The product λh then tells us something about how large the time step is compared to the rate (or speed) of the process modeled by the ODE, which makes sense.

8.1.4 Accuracy analysis

Assuming that we have a stable ODE-IVP method, we can then ask how accurate it is. For the exact problem (Eqn. (8.1)), a Taylor expansion yields (for a scalar problem to keep notation simple)

$$u_{n+1} = u_n + \frac{du}{dt}h + \frac{d^2u}{dt^2}\frac{h^2}{2} + \frac{d^3u}{dt^3}\frac{h^3}{6} + \ldots .$$

Replacing du/dt by f (according to Eqn. (8.1)) then yields

$$u_{n+1} = u_n + fh + \frac{df}{dt}\frac{h^2}{2} + \frac{d^2f}{dt^2}\frac{h^3}{6} + \ldots .$$

The function $f(u,t)$ is a function of two variables, which then implies

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}\frac{du}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}f\,,$$

where we used the fact that u is only a function of t (thus $\partial u/\partial t = du/dt$) and again the ODE itself in the last step. Repeating this process for the next higher derivative yields

$$\frac{d^2f}{dt^2} = \frac{d}{dt}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}f\right) = \frac{\partial^2 f}{\partial t^2} + 2\frac{\partial^2 f}{\partial u\partial t}f + \frac{\partial f}{\partial u}\frac{\partial f}{\partial t} + \frac{\partial^2 f}{\partial u^2}f^2 + \left(\frac{\partial f}{\partial u}\right)^2 f\,.$$

We then get that the exact u_{n+1} should be an infinite Taylor series that starts as

$$\begin{aligned} u_{n+1} = u_n + hf &+ \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}f\right) \\ &+ \frac{h^3}{6}\left(\frac{\partial^2 f}{\partial t^2} + 2\frac{\partial^2 f}{\partial u\partial t}f + \frac{\partial f}{\partial u}\frac{\partial f}{\partial t} + \frac{\partial^2 f}{\partial u^2}f^2 + \left(\frac{\partial f}{\partial u}\right)^2 f\right) \\ &+ \ldots\,, \end{aligned} \tag{8.8}$$

where all quantities on the right-hand-side should be taken from u_n and t_n.

To analyze the accuracy of any given scheme, the process is to find what the scheme defines u_{n+1} to be in terms of u_n and t_n, and then to compare it to the exact expression given by Eqn. (8.8). For example, the explicit Euler method (8.2) implies very simply that

$$u_{n+1} = u_n + hf \,,$$

which matches only the first two terms in the exact Eqn. (8.8). Therefore, the error in the explicit Euler method is $\mathcal{O}\big(h^2\big)$ over the course of a single time step. To solve the ODE over a specific interval T requires T/h time steps, which then implies that the error of explicit Euler over a fixed time T is $\mathcal{O}(h)$.

The implicit Euler method (8.3) involves $f(u_{n+1}, t_{n+1})$ which then means that we must Taylor-expand this too; we then get

$$\begin{aligned} u_{n+1} = u_n + hf(u_{n+1}, t_{n+1}) &\approx u_n + h\left[f + \frac{df}{dt}h + \frac{d^2f}{dt^2}\frac{h^2}{2} + \ldots\right] \\ &= u_n + hf + h^2\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}f\right) + \ldots \,. \end{aligned}$$

This also matches only the first two terms in the exact Eqn. (8.8), and thus implicit Euler also has an error that scales as $\mathcal{O}(h)$ over a fixed time interval.

Repeating this process for the Crank-Nicolson method shows that its Taylor expansions matches the first three terms of the exact one, and thus the error of Crank-Nicolson is of size $\mathcal{O}\big(h^2\big)$ over a fixed time interval.

8.1.5 Runge-Kutta methods

The general idea of Runge-Kutta methods is to evaluate the right-hand-side $\mathbf{f}$ (the slope of the function) at multiple points within $t \in [t_n, t_{n+1}]$ and then to define an averaged slope with which to complete the full time step. There is a whole family of Runge-Kutta methods, of different orders of accuracy with both implicit and explicit versions.

A general definition of a Runge-Kutta method is

$$\begin{aligned} \frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{h} &= \sum_{i=1}^{s} b_i \mathbf{f}_{\text{int},i} \,, \\ \mathbf{f}_{\text{int},i} &= \mathbf{f}\left(\mathbf{u}_n + h\sum_{j=1}^{i} a_{ij}\mathbf{f}_{\text{int},j}, t_n + c_i h\right) . \end{aligned} \tag{8.9}$$

The first line shows how the full time step is completed using a weighted average of s different intermediate slopes $\mathbf{f}_{\text{int},i}$, with s generally referred to as the number of “stages”. The second line defines how each intermediate slope $\mathbf{f}_{\text{int},i}$ is computed, specifically how it is defined using a weighted average of the prior $i-1$ slopes and the current ith slope as well. If the current ith slope is included (i.e., if $a_{ii} \neq 0$), the method is implicit and requires the solution of a nonlinear system of equations; if the currunt ith slope is not included (i.e., if $a_{ii} = 0$), the method is explicit.

Given this general definition, a specific Runge-Kutta method is defined by the values of all coefficients. It is customary to list all the coefficients in a "Butcher tableau" as

$$\begin{array}{c|cccc}
c_1 & a_{11} & & & \\
c_2 & a_{21} & a_{22} & & \\
\vdots & \vdots & \vdots & \ddots & \\
c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\
\hline
 & b_1 & b_2 & \dots & b_s
\end{array}$$

The most common Runge-Kutta method, often referred to as the "classic RK" method or "RK4", is the 4-stage method defined by

$$\begin{array}{c|cccc}
0 & 0 & & & \\
1/2 & 1/2 & 0 & & \\
1/2 & 0 & 1/2 & 0 & \\
1 & 0 & 0 & 1 & 0 \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}$$

This is fully explicit (zeros on the diagonal) and can be shown to be 4th order accurate, i.e., the error over a fixed interval is $\mathcal{O}\big(h^4\big)$. The stability region for RK4 is shown in Fig. 8.1. It is clearly larger than explicit Euler and also includes a fair bit of the imaginary axis. The popularity of RK4 is due to this combination a easy implementation and low computational cost (due to the explicit nature) and the rather large stability region that includes purely imaginary eigenvalues.

8.2 Boundary value problems

A boundary value problem (BVP) is composed of a differential equation and a set of boundary conditions. A common ordinary differential equation is

$$\frac{d^2u}{dx^2} = q(u,x)\,,\ x \in [0,1]\,, \tag{8.10}$$

which could for example model a one-dimensional heat conduction process in which u is the temperature, x is the coordinate, and q is a heat sink/source. The most common boundary conditions are either to specify the value of the unknown function or its derivative; these are called "Dirichlet" and "Neumann" boundary conditions, respectively. For the heat conduction example, a Dirichlet condition (e.g., $u(0) = u_L$ for a given value u_L) implies that the temperature is known at the boundary while a Neumann condition (e.g., $du/dx(1) = u'_R$ for a given value u'_R) implies that the heat flux is known instead.

An ODE boundary value problem (ODE-BVP) must necessarily have at least a second derivative in it in order to require at least one boundary condition at either side of the domain; however, it could have higher derivatives as well, and if so would need more boundary conditions at at least one of the two ends of the domain. For example, the beam bending equation in solid mechanics is of fourth order and thus requires a total of four boundary conditions, generally taken as two at either end of the domain.

Two different approaches to solving ODE-BVPs will be discussed in this book. The most general, robust and accurate method of these is the finite difference method, in which one uses the finite difference schemes developed in chapter 6 and then solves the resulting system of equations. This approach will be discussed in section 8.2.1. We will also cover an additional approach, the so-called “shooting method”, that is less robust but can be highly useful for quick tests due to its simplicity; this will be covered in section 8.2.3.

8.2.1 Finite difference method

Imagine that we want to solve Eqn. (8.10) on a uniform grid $x_j = hj, j = 0, 1, \ldots, N$ with the uniform grid-spacing $h = 1/N$. This statement actually means two different things. First, that we will describe the unknown function $u(x)$ only through its values at the grid points $u_j = u(x_j)$. This implies that we have a vector $\mathbf{u} = (u_0, u_1, \ldots, u_N)^T$ of unknowns, i.e., a total of $N + 1$ unknowns. The second and more subtle meaning is that we want to solve (or enforce) the differential equation (8.10) only at the grid points. This can only be done approximately through use of some finite difference scheme to approximate the derivative. If we choose the second-order accurate central difference scheme, this leads to

$$\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} = q_j \ , \quad j = 1, 2, \ldots, N-1 \, , \tag{8.11}$$

where $q_j = q(x_j)$ and where we have noted that this approximation works only in the “interior” grid points; it could not be applied at the boundary points since those have no neighbor to the left or right, respectively.

Now imagine that the boundary conditions for our problem are $u(0) = u_L$ (a Dirichlet condition) and $du/dx(1) = u'_R$ (a Neumann condition). Enforcing the former is straightforward, by simply requiring $u_0 = u_L$. Enforcing the latter requires us to again approximate the derivative, which in this case requires a completely left-biased scheme. The simplest choice is

$$\left.\frac{du}{dx}\right|_N \approx \frac{u_N - u_{N-1}}{h} \, .$$

Together with the $N-1$ equations describing the ODE in the interior points (Eqn. 8.11), the boundary conditions then give us a total of $N + 1$ equations which is exactly what we need. In other words, we do not enforce the ODE at the boundaries, and instead use the boundary conditions at those points. We can then summarize our system of equations as

$$\begin{aligned} u_0 &= u_L \, , \\ \frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} &= q_j \ , \quad j = 1, 2, \ldots, N-1 \, , \\ \frac{u_N - u_{N-1}}{h} &= u'_R \, . \end{aligned}$$

To solve this linear system, we assemble it in matrix-vector form as

$$\underbrace{\begin{pmatrix} 1 & 0 & \dots & & & \\ 1/h^2 & -2/h^2 & 1/h^2 & 0 & \dots & \\ 0 & 1/h^2 & -2/h^2 & 1/h^2 & 0 & \dots \\ \vdots & & \ddots & \ddots & \ddots & \\ & \dots & 0 & 1/h^2 & -2/h^2 & 1/h^2 \\ & & \dots & 0 & -1/h & 1/h \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}}_{\mathbf{u}} = \underbrace{\begin{pmatrix} u_L \\ q_1 \\ q_2 \\ \vdots \\ q_{N-1} \\ u'_R \end{pmatrix}}_{\mathbf{q}} .$$

We would then find the solution $\mathbf{u} = A^{-1}\,\mathbf{q}$ by using any linear algebra package. Note how the first and last rows of A are encoding the boundary conditions rather than the ODE, and how therefore the first and last entries in the right-hand-side vector $\mathbf{q}$ are related to the boundary conditions rather than the source term $q(x)$.

If we desired greater accuracy, we could use more accurate finite difference schemes for either the interior points and/or for the Neumann boundary condition. For example, we could use a 4th-order accurate scheme for the 2nd derivative to replace Eqn. (8.11) by

$$\frac{-u_{j-2} + 16u_{j-1} - 30u_j + 16u_{j+1} - u_{j+2}}{12h^2} = q_j \ , \quad j = 2, 3, \dots, N-2 \, .$$

This could of course not be used at grid points $j = 1$ and $j = N-1$, for which we would need to either keep the 2nd-order accurate scheme or derive a partially right-biased scheme that would utilize one neighbor on one side and maybe three on the other side. Let us for simplicity's sake choose the former option here.

Similarly, we could use a higher-order finite difference scheme to approximate the Neumann boundary condition, say

$$\left.\frac{du}{dx}\right|_N \approx \frac{u_{N-2} - 4u_{N-1} + 3u_N}{2h} \, .$$

With all this, the new version of the system of equations would have the modified matrix

$$A = \begin{pmatrix} 1 & 0 & \dots & & & \\ 1/h^2 & -2/h^2 & 1/h^2 & 0 & \dots & \\ -1/(12h^2) & 4/(3h^2) & -5/(2h^2) & 4/(3h^2) & -1/(12h^2) & \dots \\ \vdots & & \ddots & \ddots & \ddots & \\ \dots & -1/(12h^2) & 4/(3h^2) & -5/(2h^2) & 4/(3h^2) & -1/(12h^2) \\ & \dots & 0 & 1/h^2 & -2/h^2 & 1/h^2 \\ & \dots & 0 & 1/(2h) & -2/h & 3/(2h) \end{pmatrix} .$$

8.2.2 Handling nonlinearities when solving a system of equations

The focus of the previous section was on the approximation of the differential operator and the imposition of the boundary conditions. The next complication is that Eqn. (8.10) might not be linear – actually, in most applications it probably will not be linear, and we will need to find ways to deal with that. The basic problem can be condensed into the fact that all of our linear algebra tools for solving systems of equations apply to *linear* systems. Therefore, the general strategy for solving nonlinear systems of equations is to create an iterative method in which we make an initial guess for the solution $\mathbf{u}$ (let's call it $\mathbf{u}_0$), and where we then seek to iteratively improve this guess by solving a sequence of linear problems. The general strategy is identical to the one in the Newton-Raphson method for root-finding: at each iteration we create a linear problem that approximates the true problem, and we then solve that linear problem to find the "new" guess for the solution.

Nonlinearities can enter the problem in two ways in Eqn. (8.10), either through the source term $q(u, x)$ or through the introduction of a solution-dependent coefficient multiplying the differential operator. Let us deal with each case one at a time.

If we view our ODE-BVP (8.10) as a model for a heat conduction problem, then we can easily imagine a source term of form

$$q(u, x) = a(x) + b(x)u + c(x)u^4 \,,$$

where $a(x)$ could describe a fixed heat source (e.g., an electric heater with a given current through it), $b(x)u$ could describe convective cooling (e.g., the flow of cold air at given velocity, with the heat loss proportional to the temperature difference), and $c(x)u^4$ could describe radiative cooling.

In our quest for a linearized equation, we can approximate the source term as

$$q(u, x) \approx a(x) + b(x)u + c(x)\left[u_{\text{base}}^4 + 4u_{\text{base}}^3(u - u_{\text{base}})\right] \,,$$

where the term in square brackets is simply the Taylor expansion of u^4 around some known solution u_{base}. In the context of an iterative method where the solution at iteration counter n is known, we would then say that u_{base} is the known solution at iteration n and that u is the unknown solution at iteration $n + 1$.

The full linearized ODE is then (at interior points, assuming the 2nd-order accurate finite difference scheme for simplicity)

$$\frac{u_{j-1,n+1} - 2u_{j,n+1} + u_{j+1,n+1}}{h^2} = a_j + b_j u_{j,n+1} + c_j u_{j,n}^4 + 4c_j u_{j,n}^3 \left(u_{j,n+1} - u_{j,n}\right) \,,$$

where $u_{j,n}$ is the solution at grid point j and iteration n. We would then move all terms including $u_{j,n+1}$ to the left-hand-side to get

$$\frac{u_{j-1,n+1} - 2u_{j,n+1} + u_{j+1,n+1}}{h^2} - b_j u_{j,n+1} - 4c_j u_{j,n}^3 u_{j,n+1} = a_j + c_j u_{j,n}^4 - 4c_j u_{j,n}^3 u_{j,n} \,.$$

The final step is then to encode this in a matrix-vector form so we can use linear algebra tools to solve the linear system of equations at each iteration. Close inspection of the left-hand-side shows that the new matrix will be similar to the old matrix (which encoded the differential operator) but modified along the diagonal with all the linear terms from the linearized source term.

We can also do the manipulations caused by the linearization of the source term in the matrix-vector form of the equation directly. After approximating the differential operator, our equation is $A\mathbf{u} = \mathbf{q}$ with A approximating the d^2/dx^2 operator. The linearized source term can be written in matrix-vector form as

$$\mathbf{q} \approx \mathbf{a} + \mathrm{diag}(\mathbf{b})\,\mathbf{u} + \mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_{\mathrm{base}})^3\,\mathbf{u}_{\mathrm{base}} + 4\,\mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_{\mathrm{base}})^3\,(\mathbf{u} - \mathbf{u}_{\mathrm{base}})\ ,$$

where diag($\mathbf{v}$) is a diagonal matrix with the elements of the vector $\mathbf{v}$ on the diagonal. The use of all these diagonal matrices may seem strange, but is necessitated by the fact that the "point-wise" product $b_j u_j$ is *not* a vector-vector product – instead, it is

$$\mathrm{diag}(\mathbf{b})\,\mathbf{u} = \begin{pmatrix} b_0 & 0 & \dots & \\ 0 & b_1 & 0 & \dots \\ \dots & 0 & \ddots & 0 \\ & \dots & 0 & b_N \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} b_0 u_0 \\ b_1 u_1 \\ \vdots \\ b_N u_N \end{pmatrix} .$$

Similarly, when we have multiple "point-wise" multiplications (e.g., cu^4), we need to write every multiplication as the multiplication by a diagonal matrix. Finally, note that $\mathrm{diag}(\mathbf{u})^3 = \mathrm{diag}(\mathbf{u})\,\mathrm{diag}(\mathbf{u})\,\mathrm{diag}(\mathbf{u})$.

With this matrix-vector form of the linearized source term, our iterative version of the ODE is

$$A\mathbf{u}_{n+1} = \mathbf{a} + \mathrm{diag}(\mathbf{b})\,\mathbf{u}_{n+1} + \mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_n)^3\,\mathbf{u}_n + 4\,\mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_n)^3\,(\mathbf{u}_{n+1} - \mathbf{u}_n)$$

which we can re-arrange to

$$\left(A - \mathrm{diag}(\mathbf{b}) - 4\,\mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_n)^3\right)\mathbf{u}_{n+1} = \mathbf{a} + \mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_n)^3\,\mathbf{u}_n - 4\,\mathrm{diag}(\mathbf{c})\,\mathrm{diag}(\mathbf{u}_n)^3\mathbf{u}_n\,.$$

This is, of course, the same as we would have gotten by performing the linearization first and then writing it in matrix-vector form.

Nonlinearities can also enter the problem through a solution-dependent coefficient multiplying the differential operator. If we view our ODE-BVP (8.10) as a model for a heat conduction problem, imagine having a heat conductivity d on the left-hand-side that depends on the solution (temperature in our view of the problem) u. Our ODE in matrix-vector form is then

$$\mathrm{diag}(\mathbf{d}(\mathbf{u}))A\mathbf{u} = \mathbf{q}\,.$$

In this case it is the left-hand-side that is nonlinear and requires attention. A true linearization of the left-hand-side is

$$\mathrm{diag}(\mathbf{d}(\mathbf{u}))A\mathbf{u} \approx \mathrm{diag}(\mathbf{d}(\mathbf{u}_{\mathrm{base}}))A\mathbf{u} + \left.\frac{\partial \mathbf{d}}{\partial \mathbf{u}}\right|_{\mathrm{base}} (\mathbf{u} - \mathbf{u}_{\mathrm{base}})\,A\mathbf{u}_{\mathrm{base}}\,.$$

If the sensitivity in the coefficient (or "conductivity") $\partial\mathbf{d}/\partial\mathbf{u}$ is easy to find, one could implement this true linearization; however, in practice it is quite common to neglect the second term and instead solve

$$\mathrm{diag}(\mathbf{d}(\mathbf{u}_n)A\mathbf{u}_{n+1} = \mathbf{q}$$

at each iteration.

The linearization technique described above does not guarantee that the iterative solution process will converge. A different technique for stabilizing the convergence process is the idea of "under-relaxation", in essence the idea of slowing down the convergence process to avoid over-shoots. Imagine that our linearized system is

$$A_n \mathbf{u}_{n+1} = \mathbf{q}_n \,,$$

where A and $\mathbf{q}_n$ now contain all the linearized terms. The iterative process would then produce a sequence of solutions $\mathbf{u}_1, \mathbf{u}_2, \ldots$, each of which the result of solving the linearized problem. If this sequence diverges ("blows up"), one can instead try to update $\mathbf{u}$ as

$$\begin{aligned} \mathbf{u}_{n+1} &= \alpha \mathbf{v} + (1-\alpha)\mathbf{u}_n \,, \\ A_n \mathbf{v} &= \mathbf{q}_n \,, \end{aligned}$$

where $\alpha \in (0,1]$ is the under-relaxation parameter. In other words, one would solve the linear problem to find $\mathbf{v}$, but instead of taking this as the new estimate of the solution $\mathbf{u}_{n+1}$, one takes only a small portion of $\mathbf{v}$ when updating the solution. This will slow down the convergence, which is generally a bad thing – but it may be needed in cases where the solution blows up.

Under-relaxation is arguably more art than science, in the sense that one must use trial-and-error to choose the appropriate value for α. In practice, one should attempt to device a solution algorithm that does not require it, but if all else fails, under-relaxation can be useful.

8.2.3 Shooting method

The "shooting method" is an alternative way to solve ODE boundary value problems. It is not always robust or accurate, but is very simple to implement and can therefore be useful in some situations.

Imagine that we want to solve the ODE-BVP problem (8.10) with the Dirichlet boundary conditions $u(0) = u_L$ and $u(1) = u_R$. However, rather than actually solving this as a boundary value problem, we could pretend that it is an initial value problem if we assume that we have an additional boundary condition at the left boundary, in this case $du/dx(0) = u'_L$. In reality we don't know the value of u'_L until after we have found a solution, but we could cook up an algorithm like

Algorithm 8.1: the shooting method (for the ODE-BVP (8.10) with boundary conditions $u(0) = u_L$ and $u(1) = u_R$.

```
guess an initial value for $u'_L$
while not converged,
  solve the "initial value problem" with $u(0) = u_L$ and $u'(0) = u'_L$ for $u(x)$
  if $u(1) \neq u_R$, adjust $u'_L$ and try again
end while
```

The iterative process for finding the correct value of u'_L is actually a root-finding process: we are seeking the value of u'_L that makes the misfit in the right boundary condition

$f(u'_L) = u(1) - u_R$ equal to zero. It is not trivial to find the derivative of this misfit, and hence the Newton-Raphson root-finding method is generally not useful here, but we could use either the bisection and secant methods with no problems. The only difference compared to our prior uses of the root-finding methods is that each function evaluation (in the context of the root-finding method) now involves solving an ODE initial value problem.

Chapter 9

Partial differential equations

Partial differential equations (PDEs) have derivatives with respect to more than one variable. In this book we will cover only two-dimensional cases, like

$$\frac{\partial u}{\partial t} - \alpha \frac{\partial^2 u}{\partial x^2} = 0\,, \tag{9.1a}$$

$$\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0\,, \tag{9.1b}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = q\,. \tag{9.1c}$$

These three examples were chosen on purpose: they represent the simplest examples of the three different categories of PDEs. As we will see in this chapter, each category of PDE requires slightly different numerical methods.

Equation (9.1a) is a "parabolic" equation. From a physics point-of-view, it could represent an unsteady diffusion process – a hopefully familiar example would be that of unsteady heat conduction, with u representing temperature and α representing heat conductivity (actually, heat diffusivity). This equation needs an initial condition, meaning the specification of the solution at time $t = 0$ (i.e., $u(x, 0) = u_\mathrm{I}(x)$). It also needs boundary conditions at both ends, say at $x = 0$ and $x = 1$. These boundary conditions could be either of Dirichlet type with a specific value for the solution at each time (e.g., $u(0, t) = u_\mathrm{L}(t)$) or of Neumann type with a specific value for the derivative of the solution at each time (e.g., $\partial u/\partial x|_{x=1} = u'_\mathrm{R}(t)$). If we interpret this equation as unsteady heat conduction, then these boundary conditions would imply fixed temperature or fixed heat flux.

Equation (9.1b) is a "hyperbolic" equation, generally called a first-order wave equation. It describes wave propagation in a single direction, with wave speed c. It requires an initial condition and a boundary condition at only one of the two ends: if $c > 0$, the waves propagate to the right, and a boundary condition is needed on the left side of the domain but not on the right side.

Equation (9.1c) is an "elliptic" equation. Physically it could represent steady heat conduction in two dimensions, with u being the temperature. It requires boundary conditions at all sides of the domain, but no initial condition as there is no time dimension.

In chapter 8 we saw that ODEs can be of either initial-value or boundary-value type. Simplistically, we can view parabolic PDEs as having a boundary-value character in one direction and an initial-value character in the other. In contrast, hyperbolic PDEs have initial-value character in both directions, and elliptic PDEs have boundary-value character in both directions.

The most common solution method for parabolic and hyperbolic PDEs, which both have a time dimension with initial-value-problem character, is to first discretize the space dimension (x) which creates a system of ODEs and then use standard ODE-IVP solvers. This has historically been called the "method of lines". Elliptic PDEs have entirely boundary-value-problem character, and thus must be solved by using finite difference schemes to create a system of equations that is then solved using linear algebra techniques; this is entirely analogous to how we solved ODE-BVPs before.

In this book we cover only finite difference methods, but there are many other alternatives that are arguably much more common in practice, including the finite element method and the finite volume method. While the details of these methods are different, the underlying concepts are the same as covered in this book.

9.1 Parabolic problems

Consider the parabolic PDE (9.1a) with boundary and initial conditions

$$\begin{aligned} u(0,t) &= u_{\mathrm{L}}(t)\,, \\ \left.\frac{\partial u}{\partial x}\right|_{x=1} &= u'_{\mathrm{R}}(t)\,, \\ u(x,0) &= u_{\mathrm{I}}(x)\,. \end{aligned}$$

We start by discretizing this problem in space, meaning that we create a grid and use finite difference schemes to approximate the spatial derivatives at each grid point. Let us use a uniform grid $x_j = \Delta x\, j$, $j = 0, 1, \dots, N$, here, with Δx being the grid-spacing in x. Also let us use the second-order accurate finite difference scheme for the second derivative, which then produces the equation

$$\frac{du_j}{dt} = \alpha\,\frac{u_{j-1} - 2u_j + u_{j+1}}{\Delta x^2}\,, \quad j = 1, 2, \dots, N-1\,. \tag{9.2}$$

Note that this equation is valid/meaningful only at the interior grid points, away from the boundary. At the boundaries we should solve the boundary conditions rather than the PDE. The right boundary condition is of Neumann type and thus requires us to approximate the derivative using a one-sided scheme; we choose the simplest option

$$\frac{u_N - u_{N-1}}{\Delta x} = u'_{\mathrm{R}}(t)\,.$$

We now have a very subtle problem: we clearly want to write our spatially discretized problem as a system of ODE-IVPs

$$\frac{d\mathbf{u}}{dt} = \frac{\alpha}{\Delta x^2}\begin{pmatrix} \ddots & \ddots & \\ 1 & -2 & 1 \\ & \ddots & \ddots \end{pmatrix}\mathbf{u}\,,$$

but if we look carefully we realize that this will not be trivial for the boundary points. Specifically, how can we encode the boundary condition $u_0 = u_\mathrm{L}(t)$ in the context of an ODE-IVP with a time derivative? We could at best encode $du_0/dt = du_\mathrm{L}/dt$, but this would not guarantee the right value at the boundary (e.g., if u_0 was too large, there would be no mechanism for bringing it back down to the right value). Actually, when thinking about this subtle issue of how to encode the boundary condition, we at some point realize that the easiest solution is to simply not solve the boundary points as ODE-IVPs. In other words, we should define our unknown solution vector as $\mathbf{u} = (u_1, u_2, \ldots, u_{N-1})^T$, i.e., *without* the first and last points. If we do that, we can write our spatially discretized problem as

$$\frac{d}{dt}\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_j \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix}}_{\mathbf{u}} = \frac{\alpha}{\Delta x^2}\underbrace{\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -1 \end{pmatrix}}_{A}\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_j \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix}}_{\mathbf{u}} + \frac{\alpha}{\Delta x^2}\begin{pmatrix} u_\mathrm{L}(t) \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ \Delta x\, u'_\mathrm{R}(t) \end{pmatrix}. \tag{9.3}$$

Note that we used the Neumann BC on the right side to solve for u_N, which when explicitly entered into the equation for grid point $j = N-1$ yields

$$\frac{du_{N-1}}{dt} = \alpha\frac{u_{N-2} - 2u_{N-1} + u_N}{\Delta x^2} = \alpha\frac{u_{N-2} - u_{N-1} + \Delta x\, u'_\mathrm{R}(t)}{\Delta x^2}$$

which is what the last row in the ODE-IVP system (9.3) encodes.

It is useful to pause here and consider what equation (9.3) means. The first part of the right-hand-side describes how the solution itself affects the evolution in time. This is diffusion (or heat conduction) within the domain. The second part comes from the boundary condition, and shows how the boundary condition becomes a "source term" in the spatially discretized problem: the boundary condition "drives" the solution forward, and the interior dynamics respond to that driving force at the boundaries. This is the essence of a parabolic PDE.

Having discretized the problem in space and thus created a system of ODE-IVPs, we next need to choose an ODE-IVP solver. In section 8.1 we learned that different solvers have very different stability characteristics, and that stability analysis requires the eigenvalues of the A matrix in Eqn. (9.3) (more generally, the eigenvalues of the Jacobian of the right-hand-side). It is not realistic to compute the eigenvalues of A due to the large size of the matrix, which is essentially the same as the number of grid points. This is even more true to realistic PDE problems defined in 3 spatial dimensions for which one might use millions or even billions of grid points. What we want is instead an approximate estimate of what the eigenvalues of A are. The easiest way to do this is to, for the sake of this approximate estimate, assume that the problem is periodic in x; that way, Eqn. (9.2) is valid for every row in (9.3) and, more importantly, we can write the solution as a backward Fourier transform (Eqn. (4.1) in section 4.1) and get from Eqn. (9.2)

$$\begin{aligned}\sum_l \frac{d\hat{u}_l}{dt} e^{\imath 2\pi lj/N} &= \frac{\alpha}{\Delta x^2} \sum_l \hat{u}_l \left[e^{\imath 2\pi l(j-1)/N} - 2e^{\imath 2\pi lj/N} + e^{\imath 2\pi l(j+1)/N} \right] \\ &= \frac{\alpha}{\Delta x^2} \sum_l \hat{u}_l e^{\imath 2\pi lj/N} \left[e^{-\imath 2\pi l/N} - 2 + e^{\imath 2\pi l/N} \right] \\ &= \frac{\alpha}{\Delta x^2} \sum_l \hat{u}_l e^{\imath 2\pi lj/N} \left[2\cos(2\pi l/N) - 2 \right] .\end{aligned}$$

We now use the fact that $2\pi l/N = k_l \Delta x$ in the argument of the cos function since this will aid our intuitive understanding later. Orthogonality of the Fourier modes then implies that (see section 4.1)

$$\frac{d\hat{u}_l}{dt} = -\frac{\alpha}{\Delta x^2} \left[2 - 2\cos(k_l \Delta x) \right] \hat{u}_l .$$

The factor multiplying $\hat{u}_l$ on the right-hand-side is actually the eigenvalues (for every l) of the matrix A (and the eigenvectors are the Fourier modes $\exp(\imath 2\pi jl/N)$). We thus see that the eigenvalues of A for this problem are negative real values, and that they are $\leq 4\alpha/\Delta x^2$ in magnitude. Going back to the ODE-IVP solvers, we then realize that explicit Euler would work well for this problem, and that stability requires $4\alpha\Delta t/\Delta x^2 \leq 2$. Actually, since our analysis ignored the boundaries and assumed periodicity, we should view this stability bound as approximate.

The maximum (in magnitude) eigenvalue of a matrix is called the spectral radius, equal to $4\alpha/\Delta x^2$ for this problem. The spectral radius must necessarily have units of inverse time, and the $\alpha/\Delta x^2$ part is entirely due to the nature of the PDE. The factor of 4, on the other hand, comes from our choice of the second-order accurate finite difference scheme; had we chosen a more accurate scheme, this factor would have been larger.

In practice we would probably use a non-uniform grid, and the diffusivity α might not be constant. In that case we could still apply the approximate analysis using Fourier modes to find the spectral radius, and we would need to take the largest value of $\alpha/\Delta x^2$ to estimate the spectral radius; it would, of course, be even more approximate than before.

9.2 Hyperbolic problems

Consider the hyperbolic PDE (9.1b). For constant c, the analytical solution to this PDE can be written as $u(x,t) = f(x - ct)$ for any function f: this type of solution is called a "traveling wave" since it is the same shape $f(x)$ for all times but translated at speed c. This form of the solution shows clearly that, if $c > 0$, we only need a boundary condition at the left side (the "inflow" side) of the domain. We thus have the boundary and initial conditions

$$\begin{aligned}u(0,t) &= u_{\mathrm{L}}(t) , \\ u(x,0) &= u_{\mathrm{I}}(x) .\end{aligned}$$

We again follow the same "method of lines" process of first discretizing in space and then using an ODE-IVP solver to integrate the system in time. Let us again use a uniform spatial grid $x_j = \Delta x\, j$, $j = 0, 1, \ldots, N$, and let us use the second-order accurate central difference scheme for the first derivative. We need to solve the PDE at all points except $j = 0$ where we instead use the boundary condition: in contrast to the parabolic case, we must now solve the PDE at the last grid point $j = N$ since we have no boundary condition there. At this last point we can't apply the central difference scheme, and therefore must use a left-biased scheme instead. We then get the equation

$$\begin{aligned} \frac{du_j}{dt} &= -c\,\frac{u_{j+1} - u_{j-1}}{2\Delta x}\,, \quad j = 1, 2, \ldots, N-1\,, \\ \frac{du_j}{dt} &= -c\,\frac{u_j - u_{j-1}}{\Delta x}\,, \quad j = N\,. \end{aligned} \tag{9.4}$$

Following the discussion of the parabolic problem, it is easiest to define our vector of unknowns as $\mathbf{u} = (u_1, u_2, \ldots, u_N)^T$, i.e., to contain the solution at all grid points except for where we have a boundary condition (so this vector has one additional element compared to the parabolic case). We then get the system of ODE-IVPs

$$\frac{d}{dt}\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_j \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}}_{\mathbf{u}} = -\frac{c}{\Delta x}\underbrace{\begin{pmatrix} 0 & 0.5 & & & & \\ -0.5 & 0 & 0.5 & & & \\ & \ddots & \ddots & & & \\ & -0.5 & 0 & 0.5 & & \\ & & \ddots & \ddots & & \\ & & & -0.5 & 0 & 0.5 \\ & & & & -1 & 1 \end{pmatrix}}_{A}\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_j \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}}_{\mathbf{u}} + \frac{c}{\Delta x}\begin{pmatrix} 0.5u_{\mathrm{L}}(t) \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}. \tag{9.5}$$

Just like for the parabolic problem, the boundary condition has become (after discretization in space) a source term that forces the solution near the boundary, which the interior dynamics (described by the PDE) then propagates into the domain.

We again face the same problem of having to estimate the eigenvalues of the A matrix in order to first choose a suitable ODE-IVP solver and then decide on the largest stable time step Δt. Exactly as before, this is best accomplished in an approximate manner, by assuming (just for the sake of this estimation) that we have a periodic domain for which we can apply the discrete Fourier transform. Orthogonality of the Fourier modes then leads to, for the central difference scheme,

$$\frac{d\hat{u}_l}{dt} = -\frac{c}{\Delta x}\hat{u}_l\left[\frac{e^{\imath k_l \Delta x} - e^{-\imath k_l \Delta x}}{2}\right] = -\imath\frac{c}{\Delta x}\sin(k_l \Delta x)\,\hat{u}_l\,.$$

Note that the factor multiplying $\hat{u}_l$ (i.e., the eigenvalue for the periodic domain case) is now purely imaginary, with a maximum magnitude of $c/\Delta x$. This is very important information, because it tells us that explicit Euler will blow up for this problem! Actually, the same is true for two-stage Runge-Kutta, as both of those schemes are unstable for purely imaginary eigenvalues. In contrast, three- and four-stage Runge-Kutta methods do contain parts of the imaginary axis in their stability regions, and they are therefore very suitable to this hyperbolic problem. If we use RK4, say, we would need $c\Delta t/\Delta x \lesssim 2.8$ (where the stability region intersects the imaginary axis), which we can use to choose a stable time step.

An alternative way to achieve a stable solution of this hyperbolic problem is to use a left-biased finite difference scheme. Imagine that we used

$$\frac{du_j}{dt} = -c\frac{u_j - u_{j-1}}{\Delta x}$$

throughout the domain. The stability analysis would then end up with

$$\frac{d\hat{u}_l}{dt} = -\frac{c}{\Delta x}\hat{u}_l\left[1 - e^{-\imath k_l \Delta x}\right] = -\frac{c}{\Delta x}\left[1 - \cos(k_l\Delta x) + \imath\sin(k_l\Delta x)\right]\hat{u}_l\,.$$

The eigenvalues (the factor multiplying $\hat{u}_l$ on the right-hand-side) now have a negative real part! If one plots them for the range of allowable $k\Delta x \in [-\pi, \pi]$ one finds that the eigenvalues lie on a circle in the negative half-plane. Provided that $c\Delta t/\Delta x \leq 1$, all eigenvalues actually fall inside the stability region of the explicit Euler scheme, which implies that we could now use this method.

This example illustrates something very important: the characteristics of a spatially discretized hyperbolic PDE depend very strongly on the specific finite difference scheme chosen to approximate the spatial derivative. While the specific choice of finite difference scheme changes the multiplicative factor in the eigenvalues for parabolic problems, it has a much more dramatic qualitative effect of changing the nature of the eigenvalues for hyperbolic problems.

The idea of using a left-biased scheme for this problem is more generally called "upwinding". If we solve the same problem but with $c < 0$ instead, our left-biased scheme would produce eigenvalues with positive real part, implying divergence; we would then need to use a right-biased scheme instead to have a stable method. In other words, we need to bias our finite difference scheme towards the "upstream" direction, or the direction from which the waves are traveling. We actually don't need a fully upwind-biased scheme to achieve this effect, we just need to have a bit of bias towards the upstream. For example, consider the finite difference scheme

$$\frac{du_j}{dt} = -c\frac{0.4u_{j+1} + 0.2u_j - 0.6u_{j-1}}{\Delta x}\,,$$

which places a bit more weight on the upstream point versus the downstream point (0.6 versus 0.4). For this scheme, the stability analysis ends up with

$$\begin{aligned}\frac{d\hat{u}_l}{dt} &= -\frac{c}{\Delta x}\hat{u}_l\left[0.4e^{\imath k_l\Delta x} + 0.2 - 0.6e^{-\imath k_l \Delta x}\right]\\ &= -\frac{c}{\Delta x}\left[0.4\cos(k_l\Delta x) + 0.4\imath\sin(k_l\Delta x) + 0.2 - 0.6\cos(k_l\Delta x) + 0.6\imath\sin(k_l\Delta x)\right]\hat{u}_l\\ &= -\frac{c}{\Delta x}\left[0.2 - 0.2\cos(k_l\Delta x) + \imath\sin(k_l\Delta x)\right]\hat{u}_l\,.\end{aligned}$$

This also adds some negative real part to the eigenvalues, but with five times lower magnitude. Explicit Euler would still be stable, but a more detailed analysis reveals that it would require a five times smaller time step Δt.

9.3 Elliptic problems

Consider the elliptic PDE (9.1c), which requires boundary conditions on all boundaries. These boundary conditions could be of either Dirichlet or Neumann type; for example, we could take

$$\begin{aligned} u(0,y) &= u_{\mathrm{L}}(y)\,, \\ \left.\frac{\partial u}{\partial x}\right|_{x=1} &= u'_{\mathrm{R}}(y)\,, \\ u(x,0) &= u_{\mathrm{B}}(x)\,, \\ \left.\frac{\partial u}{\partial x}\right|_{y=1} &= u'_{\mathrm{T}}(x)\,. \end{aligned}$$

Elliptic problems are like boundary value problems in both directions, and we must therefore discretize both directions directly using finite difference schemes. To do this, we must first create a grid of points. For simplicity, let us use a uniform and perfectly Cartesian grid here, with

$$\begin{aligned} x_{i,j} &= \Delta x\, i\,,\ \ i = 0,1,\ldots,N_x\,, \\ y_{i,j} &= \Delta y\, j\,,\ \ j = 0,1,\ldots,N_y\,. \end{aligned}$$

Using the second-order accurate central difference scheme in both directions, the discretized problem at grid point (i,j) becomes

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = q_{i,j}\,.$$

At the boundary points we need to enforce the boundary conditions rather than solve the PDE. Unlike for the time-evolving parabolic and hyperbolic problems, there is no difficulty in enforcing the boundary conditions here; we can therefore take our vector of unknowns $\mathbf{u}$ to include the boundary points. The only problem is a more practical one: we need to create a *vector* of unknowns $\mathbf{u}$ from the two-dimensional grid of unknowns $u_{i,j}, i = 0,1,\ldots,N_x, j = 0,1,\ldots,N_y$. This is done by stacking the points either by "row" (fixed j) or by "column" (fixed i). For example, we can choose to view i is the fastest varying index, in which case we define

$$\mathbf{u} = (u_{0,0}, u_{1,0}, u_{2,0}, \ldots, u_{N_x,0}, u_{0,1}, u_{1,1}, \ldots, u_{N_x,1}, \ldots, u_{N_x-1,N_y}, u_{N_x,N_y})^T\,.$$

If we write our discretized problem as $A\mathbf{u} = \mathbf{q}$, a typical row in A corresponding to an interior grid point is then (with zeros in every other location)

$$\begin{pmatrix} \cdots & \frac{1}{\Delta y^2} & \cdots & \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} - \frac{2}{\Delta y^2} & \frac{1}{\Delta x^2} & \cdots & \frac{1}{\Delta y^2} & \cdots \end{pmatrix}\,.$$

Note how the matrix has 5 non-zero entries in every row, corresponding to the 5 grid points involved in the finite difference stencil at each point.

The rows corresponding to the boundary points must encode the boundary conditions. As before, Dirichlet conditions are encoded by a 1 in the correct position and zeros elsewhere, with the boundary value in the $\mathbf{q}$ vector on the right-hand-side. Also as before, Neumann conditions are encoded by choosing a fully one-sided first derivative scheme. After encoding the boundary conditions, the system can be solved using linear algebra techniques.

Bibliography

Bendat, J. S. & Piersol, A. G. 2010 *Random data: analysis and measurement procedures*. Wiley & Sons.

Chapra, S. C. & Canale, R. P. 2015 *Numerical methods for engineers*. McGraw-Hill.

Heath, M. T. 2018 *Scientific computing: an introductory survey*. SIAM.

Russo, S. & Luchini, P. 2017 A fast algorithm for the estimation of statistical error in DNS (or experimental) time averages. *J. Comput. Phys.* **347**, 328–340.

Sivia, D. S. 2006 *Data analysis: a Bayesian tutorial*. Oxford University Press.

Trefethen, L. N. & Bau III, D. 1997 *Numerical linear algebra*. SIAM.