A Brief Introduction to Numerical Methods in Physics

Julius B. Kirkegaard

December 17, 2021

Contents

Introduction

These notes are meant to serve as an *accessible* introduction to a *broad* range of computational methods often employed in physics. *Understanding* is valued over full mathematical details and in-depth concepts.

Once you have read this text you should

1. have a good overview of what numerical methods exists, and be able to choose an appropriate method when faced with a challenging problem.
2. have a basic understanding of most methods. At least enough to get you started writing some code, and definitely enough for you to be comfortable reading more detailed descriptions of the method elsewhere.

The text is meant to be accompanied by a course that teaches hand-on how to apply these methods in a specific programming environment.

Numerous methods will be presented to tackle a diverse set of problems. It is not expected that you will master all these methods, nor even ever use many of them. Instead, it is expected that you will understand most of them and employ only some of them. Nonetheless, even if you never use a specific method, there is a great deal of value in having an overview of what methods exist. If you do not know what exists, how would know what to look for? Further, in the academic world one often find oneself in a situation where some obscure technique pops up, in which case it is very useful to be able to put this in a context of previously learned material. It will make reading other people's code easier and it will make listening to talks more interesting.

First and foremost though, the aim of these notes is to teach you methods that you will use. Some of these methods could be become the hammer that you will use to solve most problems. The aim is to give you enough understanding to implement simple versions of all methods, and provide enough of a background to be able to take a deep dive into the specifics if needed.

The notes are kept code-free. We will discuss the mathematics needed to implemented the methods presented and will assume only basic functionality of the programming language of your choice. Many methods presented will exist in some form implemented in libraries that can be downloaded and used. If these libraries are of high quality, we naturally recommend to use these. In this case understanding the underlying methods become important primarily to understand how to tune the methods and when to expect things to go wrong, or, naturally, to help when the code needs to be adapted to suit a specific need.

Numerical Differentiation

Physical laws are often defined by differential equations, and the solution of these will be the main subject of this text. It is therefore crucial that we understand how to do differentiation of functions numerically.

This chapter will go over a few methods for numerical differentiation and discuss the accuracy of these methods.

2.1 First Order Derivatives

Consider a function f of a real number x on some interval $[a, b]$. This could an analytically defined function such as $\sin x$, $\exp(x)$, x^2, and so on, or one defined more indirectly but nonetheless computable numerically. Such a function naturally accepts any value of x, for instance 1.0, π, $1/3$, and so on. In order to able to store a representation of this function on a computer, however, we are forced to only store the value of the function at a finite number of values. The standard approach is to discretise the interval of interest by some small number Δx[1], and consider the function only at these points:

$$x \in \{a, a + \Delta x,\ a + 2\Delta x,\ a + 3\Delta x,\ \cdots,\ b - \Delta x,\ b\}. \tag{2.1}$$

[1] Note that many texts use h in place of Δx.

To numerically find the derivative of $f(x)$ at one of these points, we could then for instance do:

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}. \tag{2.2}$$

This will only be truely equal in the limit of infinitesimally small Δx, but will be a good approximation also for reasonable small Δx. However, Eq. (2.2) is not the only possible choice we could take to approximate the derivative. We could also do

$$f'(x) \approx \frac{f(x) - f(x-\Delta x)}{\Delta x}. \tag{2.3}$$

Which one of these two are the best? Intuitively, we expect both of these to give equally good approximations.

In fact, there are many more choices. We could do something crazy like

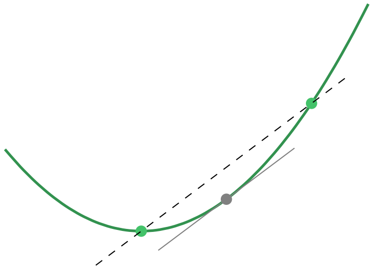
$$f'(x) \approx \frac{f(x+2\Delta x) - f(x)}{2\Delta x}, \tag{2.4}$$

which indeed also tends to $f'(x)$ as $\Delta x \to 0$. For finite Δx, however, this is a worse approximation than Eq. (2.2) and (2.3).

Eq. (2.2) and (2.3) are known as the *forward* and *backward* (or *right* and *left*) approximations to the derivative. We can actually do better than this (we will define what 'better' means in a second) by using the so-called *central* derivative

$$f'(x) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}. \tag{2.5}$$

That this is a good derivative approximation is easily appreciated from an illustration:



Consider using the forward or backward derivative with the same Δx on the illustrated function: this would give a much worse approximation of the derivative.

All of these formulas are, for good reason, called *finite difference* expressions. To express how good an approximation is, we consider what happens when applying them to the Taylor expansions of functions. Smooth functions can (locally) be approximated by their Taylor expansion

$$
\begin{aligned}
f(x - x_0) &= \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n && (2.6)\\
&= f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 + \cdots
\end{aligned}
$$

Let us try and use our finite difference formulas on these Taylor expansions. First we use the forward scheme of Eq.(2.2). Without loss of generality we evaluate the derivative for $x = 0$:

$$f'(0) \approx \frac{f(\Delta x) - f(0)}{\Delta x} = f'(0) + \frac{1}{2} f''(0)\Delta x + \cdots \tag{2.7}$$

We find that the error of the approximation has a term that grows proportional to Δx. The next error term grows like Δx^2, but for small Δx this will be much smaller, and the following terms even smaller than that. For this reason one writes

Forward Derivative

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x). \tag{2.8}$$

The notation $\mathcal{O}(\Delta x)$ signifies that the error grows like $\sim \Delta x$. If we do the same calculation for the central derivative we find

Central Derivative

$$f'(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2). \tag{2.9}$$

It is in this sense that the central derivative is better: the error grows like Δx^2, which for small Δx will be a lot smaller than Δx.

2.2 Higher Order Derivatives

We will also need to be able to take higher order derivatives. A simple way to do this is to take single derivatives multiple times.

For instance, by using Eq. (2.9) on f' we have

$$f''(x) = \frac{f'(x+\Delta x) - f'(x-\Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2).$$

And then using the formula for $x + \Delta x$ and $x - \Delta x$ we have

$$\begin{aligned} f'(x+\Delta x) &= \frac{f(x+2\Delta x) - f(x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \\ f'(x-\Delta x) &= \frac{f(x) - f(x-2\Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \end{aligned}$$

Combining these we find

$$f''(x) = \frac{f(x+2\Delta x) + f(x-2\Delta x) - 2f(x)}{4\Delta x^2} + \mathcal{O}(\Delta x), \qquad (2.10)$$

which is a finite difference approximation for the second derivative. Note that we used $\mathcal{O}(\Delta x^2)/\Delta x = \mathcal{O}(\Delta x)$, and so this approximation is only guaranteed to be accurate to within Δx. It is in fact better than this, but still worse than the more natural approximation:

Central Second Derivative

$$f''(x) = \frac{f(x+\Delta x) + f(x-\Delta x) - 2f(x)}{\Delta x^2} + \mathcal{O}(\Delta x^2). \qquad (2.11)$$

In a similar fashion, for a third order derivative one has

Central Third Derivative

$$f'''(x) = \frac{1}{2\Delta x^3}\Big[f(x+2\Delta x) - 2f(x+\Delta x) \\ +2f(x-\Delta x) - f(x-2\Delta x)\Big] + \mathcal{O}(\Delta x^2). \tag{2.12}$$

Finite difference schemes can be neatly described by just giving their *coefficients*. For instance, the third order derivative scheme above can be described simply by the coefficients $\{-\frac{1}{2}, 1, 0, -1, \frac{1}{2}\}$, corresponding to the factors of $f(x-2\Delta x)$, $f(x-\Delta x)$, $f(x)$, $f(x+\Delta x)$, and $f(x+2\Delta x)$, respectively.

In this simple way we can write down the schemes in a neat table. Here are the first few central finite difference coefficients that have second order accuracy, i.e. those for which the error grows like $\mathcal{O}(\Delta x^2)$:

Central Difference Coefficients with Accuracy $\mathcal{O}(\Delta x^2)$

	$-2\Delta x$	$-\Delta x$	0	Δx	$2\Delta x$
$f'(x)$	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0
$f''(x)$	0	1	-2	1	0
$f'''(x)$	$-\frac{1}{2}$	1	0	-1	$\frac{1}{2}$
$f''''(x)$	1	-4	6	-4	1

Note that the schemes all sum to zero and are symmetric for even derivatives and anti-symmetric for odd derivatives. Why must this be the case?

We will also give a few schemes for forward derivatives

Forward Difference Coefficients with Accuracy $O(\Delta x^2)$

	0	Δx	$2\Delta x$	$3\Delta x$	$4\Delta x$
$f'(x)$	$-\frac{3}{2}$	2	$-\frac{1}{2}$	0	0
$f''(x)$	2	-5	4	-1	0
$f'''(x)$	$-\frac{5}{2}$	9	-12	7	$-\frac{3}{2}$

Backward derivatives are found by inverting the above and flipping the signs for odd derivatives.

To use these formulas for a specific value of Δx, the coefficients should be divided by Δx^d, where d is the derivative being taken.

2.3 Deriving Schemes

In the previous section we presented a method to evaluate the accuracy of a finite difference schemes: apply the scheme to the Taylor expansion of a function and check how the error grows with Δx. We should be able to reverse engineer this approach in order to derive schemes that have a required accuracy.

In general we will find that the more points we allow evaluation at, the higher the accuracy we can attain. The set of evaluation points are called the *stencil*. In the central difference scheme, for example, we used the stencil $\{x-2\Delta x, x-\Delta x, x, x+\Delta x, x+2\Delta x\}$.

This is a regularly spaced stencil. Sometimes, there is the need for irregularly spaced points, and so we will need a scheme for general points such as $\{x_1, x_2, x_3, x_4, x_5\}$. In other words, we want a formula for the d'th derivative of the form:

$$f^{(d)}(x) \approx a_1 f(x_1) + a_2 f(x_2) + a_3 f(x_3) + a_4 f(x_4) + a_5 f(x_5),$$

where the a's are the finite difference coefficients for this custom stencil, which in general will depend on x.

The derivation of the formula is quite simple: we just need to ensure that the terms of the Taylor expansion equal zero except at the derivative we are trying to calculate, where instead we need to correct for the factorial. We skip a step-step derivation and simply state that the correct coefficients for evaluation at $x = 0$ are found by solving the following linear equation[2]

General Formula for 5-Point Finite Difference Coefficients

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 \\ x_1^3 & x_2^3 & x_3^3 & x_4^3 & x_5^3 \\ x_1^4 & x_2^4 & x_3^4 & x_4^4 & x_5^4 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = d! \begin{pmatrix} \delta_{0,d} \\ \delta_{1,d} \\ \delta_{2,d} \\ \delta_{3,d} \\ \delta_{4,d} \end{pmatrix}. \tag{2.13}$$

Here δ is the Kronecker delta. The formula generalises straightforwardly to smaller or larger stencils. The resulting scheme will

[2] If you $x \neq 0$, all x_i should be replaced by $(x_i - x)$.

have an accuracy of at least $\mathcal{O}(\Delta x^{N-d})$, where N is the number of stencil points, and Δx is representative of the distance between them. Central schemes will be one order higher. Note that point of evaluation does not need to be one of the stencil points; the formula can interpolate/extrapolate at any value.

Example

To find the third derivative scheme for the regular stencil $\{x - 2\Delta x, x - \Delta x, x, x + \Delta x, x + 2\Delta x\}$, one would solve

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2\Delta x & -\Delta x & 0 & \Delta x & 2\Delta x \\ (-2\Delta x)^2 & (-\Delta x)^2 & 0 & \Delta x^2 & (2\Delta x)^2 \\ (-2\Delta x)^3 & (-\Delta x)^3 & 0 & \Delta x^3 & (2\Delta x)^3 \\ (-2\Delta x)^4 & (-\Delta x)^4 & 0 & \Delta x^4 & (2\Delta x)^4 \end{pmatrix} \begin{pmatrix} a_{-2} \\ a_{-1} \\ a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 6 \\ 0 \end{pmatrix}.$$

With this formula we can calculate schemes for any order of derivative for any stencil (as long as $N > d$). To calculate the schemes presented in the tables above, you can set $\Delta x = 1$.

2.4 Machine Precision

You might wonder why we need to develop all these different schemes. Even a method that has an error of the order $\mathcal{O}(\Delta x)$ should still be good enough if we simply choose Δx small enough, right? In theory, this is correct, but in practice it is not.

Since computers cannot store numbers with infinite precision there are limits to the above suggestion. An immediate problem is that Δx simply cannot be chosen to be arbitrarily small. Using "single-precision floating points"[3], the smallest representable number is about 10^{-38}. Using double-precision we can get down to around 10^{-324}. This is indeed quite small, and not the source of the real problem.

When we do calculations on a computer, the important factor in maintaining precision is how big the gaps are between the numbers we can represent. This is called the *machine epsilon*. Using double-precision, for instance, the first number we can represent which is bigger than 1 is

$$1 + \epsilon_M = 1 + 2^{-52} \approx 1 + 2.2 \cdot 10^{-16}. \tag{2.14}$$

So what happens if you ask a computer to calculate $1 + 10^{-16}$? It will simply return 1.[4] This is called a *round-off* error, for obvious reasons. In general, the gap between numbers that are of magnitude N will be $\sim \epsilon_M N$. In this way, the *relative* error of doing a calculation on a computer will for double-precision floating points be of order ϵ_M.

Consider, for example, a situation where we need to estimate a derivative with absolute precision of some tolerance δ. Using

[3] Single-precision floats are often written as FLOAT32 or simply FLOAT and are real numbers stored on the computer using 32 bits, i.e. 32 zeros and ones. Double-precision floats are often written as FLOAT64 or DOUBLE and are real numbers stored using 64 bits.

[4] Likewise, trying to calculate $1 + 10^{-16} + 10^{-16} + 10^{-16}$ will also return 1, but $1 + 10^{-15}$ will return a number larger than 1.

a scheme that has error of order $O(\Delta x)$ we thus have to choose $\Delta x \sim \delta$. We can do a simple calculation where we take round-off errors of $f(x)$ into account:[5]

$$\frac{f(x+\Delta x) - f(x) \pm \epsilon_M |f(x)|}{\Delta x} = f'(x) + O(\Delta x) \pm \frac{\epsilon_M |f(x)|}{\Delta x}.$$

Here we explicitly see that on a computer, finite difference schemes will have two sources of errors: *truncation errors* that are due to our schemes not being precise, and *rounding errors* that are due to the finite number representation of computers. In our example, the truncation error would be $\sim \delta$, and the rounding error $\sim \epsilon_M/\delta$ if $|f(x)| \approx 1$.

Had we instead used the central scheme we would have

$$\frac{f(x+\Delta x) - f(x-\Delta x) \pm \epsilon_M |f(x)|}{2\Delta x} = f'(x) + O(\Delta x^2) \pm \frac{\epsilon_M |f(x)|}{2\Delta x}.$$

But now in order to reach an error of size δ, we only need to choose $\Delta x \sim \sqrt{\delta}$. With this choice, the truncation error remains the same, but the rounding error is now only $\epsilon_M / 2\sqrt{\delta}$.

This is why higher-order schemes are useful: they allow us to use a larger Δx, which minimises round-off errors. For a derivative of order d, the round-off error becomes of size $\sim \epsilon_M / \Delta x^d$. In this way, rounding errors become increasingly problematic for higher order derivatives. Spectral methods that we present in the next chapter avoid this problem to a large degree.

[5] We use the fact that $f(x)$ (and $f(x+\Delta x)$) will have a round-off error of approximately $|f(x)|\epsilon_M$.

In our analysis above we considered the round-off error of $f(x)$ and $f(x+\Delta x)$. However, there could also be a round-off error for $x+\Delta x$. This error leads to

$$\frac{f(x+\Delta x \pm \epsilon_M|x|) - f(x)}{\Delta x} = \left(1 \pm \frac{\epsilon_M|x|}{\Delta x}\right) f'(x) + \mathcal{O}(\Delta x). \quad (2.15)$$

However, this type of round-off error can actually be avoided by choosing the grid of x and spacing Δx in such a way that rounding never happens. If high precision is your aim, it is recommended to think about this.

In the following chapters we will consider differential equations. For these, higher-order schemes are critical not just for precision, but also for the speed of computation, since the larger the Δx or Δt, the fewer calculations we need to do.

2.5 Spectral Methods

In the above sections we have introduced finite difference schemes for taking numerical derivatives. The approximation of a finite difference scheme lies in approximating the derivative operator. A contrasting approach is to instead make an analytical approximation of the function and then take exact derivatives of this approximation. Such approaches are called *spectral methods*.

The simplest example of spectral methods uses Fourier series. As such let us consider a periodic function f defined on the interval $[0, 2\pi)$. Instead of storing the values of the function at a finite number of points, we could instead store a finite number of coefficients in a Fourier series. We could for instance store $2N + 1$ coefficients[6] $\{c_k\}$ and approximate

$$f(x) \approx \sum_{k=-N}^{N} c_k \, e^{ikx}. \tag{2.16}$$

Now that the true function f is approximated by the Fourier series, we can approximate derivatives of the true function by exact derivatives of Fourier series. Concretely,

$$f^{(d)}(x) \approx \sum_{k=-N}^{N} (ik)^d \, c_k \, e^{ikx}. \tag{2.17}$$

So the d'th derivative of a function represented by coefficients $\{c_k\}$ is approximated by the function represented by coefficients $\{(ik)^d c_k\}$.

If the function f is stored at regularly spaced points such as

$$x \in \{0, \Delta x, 2\Delta x, 3\Delta x, \cdots, 2\pi - \Delta x\}, \tag{2.18}$$

spectral methods can still be used. In this case, we simply use a discrete Fourier transform. Most programming language offer a very fast version called the fast fourier transform ("fft"). All in all, the d'th derivative can then be taken as

[6] If the function is real, i.e. not complex, fewer coefficients need to be stored, since then $c_{-k} = c_k^*$.

Spectral Derivative using Fast Fourier Transform

$$\boldsymbol{f}^{(d)} \approx \operatorname{ifft}\left((i\boldsymbol{k})^d \operatorname{fft}(\boldsymbol{f})\right) \tag{2.19}$$

Here we assume that $f(x)$ is stored as $\boldsymbol{f} = [f_0, f_1, f_2, \cdots, f_n]$, and 'fft' and 'ifft' are the fast fourier transform and inverse fast fourier transform, respectively.

Fourier transforming leads to complex numbers, which we then transform (here by multiplying by $(ik)^d$) and transform back to real numbers. These intermediary complex numbers can lead to spurious errors for real numbers. To avoid such problems most fft-libraries will have functions specifically designed for real-valued functions. These are typically called 'rfft' and 'irfft', and we recommend their use.[7]

For problems that require high-order numerical derivatives, spectral methods are often the preferred approach. The exception to this rule is when the function of interest has discontinuities. In contrast to finite differences, spectral method are non-local. A discontinuity at some x can therefore influence the result even far away from x. The method only works when the function is smooth everywhere.

[7] If you do not have access to such functions, it is best practice to zero out the so-called Nyquist frequency after Fourier transforming. Also note that this will depend on whether you are using and even or odd number of grid points. We mention these details so that you are aware of such issues, but skip a full discussion of them here.

Spectral methods can be combined with other approaches by using fast fourier transforms to switch between real space and frequency space appropriately. Such approaches are often called *pseudo*-spectral methods.

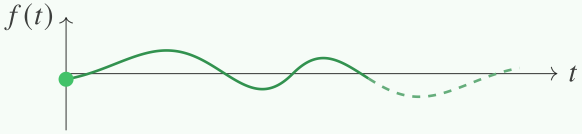
Spectral methods also exist for non-periodic domains, but the formulas differ as different basis functions need to be used. These will typically be slightly more involved to implement as conversion between function values and basis function coefficients is not necessarily pre-implemented in your programming language of choice in contrast to Fourier series.

Ordinary Differential Equations

Numerical problems involving Ordinary Differential Equations (ODEs) typically come in two forms distinguished by their boundary conditions: initial-value problems and boundary-value problems.

Initial-value problems are typically equations in time, where the value of the function is known at $t = 0$.

Initial-Value Problem



Find $\boldsymbol{f}(t)$ for $t \in [0, T]$ such that

$$\boldsymbol{f}'(t) = \boldsymbol{F}(\boldsymbol{f}(t), t) \tag{3.1}$$

and $\boldsymbol{f}(0) = \boldsymbol{f}_0$.

Here $\boldsymbol{F}$ is any (non-linear) function and $\boldsymbol{f}$ is the function to be found, both of which could be multi-dimensional. $\boldsymbol{f}_0$ is a constant vector containing the initial value of the function.

The definition includes higher-order derivative problems as these can always be recast as a set of first order equations. For example, the initial value problem

$$\begin{cases} f''(t) = F(f(t), t) \\ f(0) = \alpha \\ f'(0) = \beta \end{cases} \tag{3.2}$$

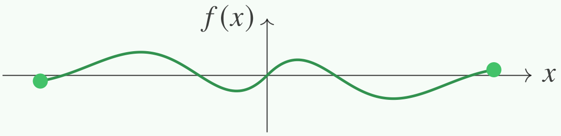
is equivalent to

$$\begin{cases} f_1'(t) = f_2(t) \\ f_2'(t) = F(f_1(t), t) \\ f_1(0) = \alpha \\ f_2(0) = \beta \end{cases} \tag{3.3}$$

Likewise, any ODE whose highest derivative is of order d can be recast as d first-order equations. The problem specification must then contain exactly d boundary conditions.

In contrast, boundary-value problems are typically problems in space for which boundary conditions are specified at the domain borders:

Boundary-Value Problem



Find $\boldsymbol{f}(x)$ for $x \in [a, b]$ such that

$$\boldsymbol{f}'(x) = \boldsymbol{F}(\boldsymbol{f}(x), x) \tag{3.4}$$

and $n = \dim(\boldsymbol{f})$ conditions of the form $f_j(b_i) = \alpha_i$ are satisfied, where b_i is equal to a or b.

An example of a boundary value problem is

$$\begin{cases} f_1'(x) = f_2(x) \\ f_2'(x) = F(f_1(x), x) \\ f_1(a) = \alpha \\ f_1(b) = \beta \end{cases} \tag{3.5}$$

At first sight the difference between initial-value and boundary-value problems might seem insignificant. This not the case, however. Initial-value problems can typically be solved step-by-step in the sense that knowing e.g. $f_1(0)$ allows you to calculate $f_1(\Delta t)$, which in turn allows you to calculate $f_1(2\Delta t)$ and so on. This is because all f_i's are known for the same time points. In contrast, for boundary-value problems, knowing $f_1(a)$ does not allow one to calculate $f_1(a+\Delta x)$ directly since it could be the case that f_2 is only known at $f_2(b)$, but its value is needed at $x = a$. These aspects, and how they are dealt with, will become clear in following sections.

3.1 Initial-Value Problems

Choosing a step-size Δt to discretise time by, we can use the forward scheme of Eq. (2.8) to immediately give a method for solving initial-value problems:

The Euler Method

$$\boldsymbol{f}(t+\Delta t) = \boldsymbol{f}(t) + \boldsymbol{F}(\boldsymbol{f}(t), t)\,\Delta t \tag{3.6}$$

We know from Eq. (2.8) that the forward scheme has error of size $\mathcal{O}(\Delta t)$, which means that each step in the Euler method has an error of size $\mathcal{O}(\Delta t^2)$[1]. In order to integrate the equation from $t = 0$ all the way to $t = T$ using a step of size Δt, we will need to do the above approximation $n \approx T/\Delta t$ times. This means that the error of the final term $\boldsymbol{f}(T)$ will be of order $T/\Delta t \times \mathcal{O}(\Delta t^2) = \mathcal{O}(\Delta t)$. Although the method is exceedingly simple to implement, this large error makes it unfit for many applications unless very small Δt are used.

Example

Let us solve

$$f'(t) = \sqrt{f(t)}, \quad f(0) = 1 \tag{3.7}$$

using the Euler method with $\Delta t = 0.1$.
The first time step gives us

$$\begin{aligned} f(\Delta t) = f(0.1) &\approx f(0) + \sqrt{f(0)}\,\Delta t \\ &= 1 + 0.1 = 1.1. \end{aligned} \tag{3.8}$$

[1] Eq. (2.8) gives $f'(t) = \frac{f(t+\Delta t)-f(t)}{\Delta t} + \mathcal{O}(\Delta t)$ which we multiply by Δt to obtain the Euler Method. The multiplication is the reason the error becomes $\mathcal{O}(\Delta t^2)$.

And the next

$$\begin{aligned} f(2\Delta t) = f(0.2) &\approx f(0.1) + \sqrt{f(0.1)}\,\Delta t \\ &= 1.1 + \sqrt{1.1}\cdot 0.1 \approx 1.205. \end{aligned}$$

And so we continue, easily implemented in a loop. To get an accurate solution we should use a smaller Δt.

In physical problems one often needs to solve versions of Newton's or Hamilton's equations. For these equations we can define spatial variables x and velocity/momentum variables v, such that the equations can be written

$$x'(t) = v(t), \tag{3.9}$$

$$v'(t) = a(x(t)), \tag{3.10}$$

where $a(x)$ is the acceleration term. We have assumed a conservative system. In this specific situation there is a small variation of the Euler method which has some very nice features and is almost as easy to implement:

Verlet/Leapfrog Integration

For each time step do

$$\begin{aligned} x_{\Delta t/2} &= x(t) + \frac{1}{2}v(t)\Delta t \\ v(t+\Delta t) &= v(t) + a(x_{\Delta t/2})\Delta t \\ x(t+\Delta t) &= x_{\Delta t/2} + \frac{1}{2}v(t+\Delta t)\Delta t \end{aligned} \tag{3.11}$$

The error of this method is only $\mathcal{O}(\Delta t^3)$ per time step, which is a big improvement over the Euler Method. This is despite being computationally as cheap as the Euler method: $a(x)$, which is typically the most computationally expensive term to calculate, is only evaluated once per time step for both methods.

A conservative system of equations will conserve energy, but we cannot in general guarantee this from numerical approximations. The Verlet (or Leap-Frog) method is a so-called *symplectic* method, which precisely ensures that the (time-averaged) energy is conserved. This feature can be crucial in many physical simulations.

3.1.1 Runge–Kutta Methods

For finite difference schemes we found that using larger stencils allowed us a better estimation of the derivatives. Runge–Kutta methods use exactly the same idea, but for initial-value problems. The Euler method has error $\mathcal{O}(\Delta t^2)$ per time step. A simple Runge–Kutta method to improve on this is

Second Order Runge–Kutta (Midpoint Method)

For each time step update

$$\begin{aligned} k_1 &= F(f(t), t) \\ k_2 &= F(f(t) + \frac{1}{2}k_1\Delta t, t + \frac{1}{2}\Delta t) \\ f(t+\Delta t) &= f(t) + k_2\Delta t \end{aligned} \tag{3.12}$$

This method is strikingly similar to Verlet integration in that we do a half-Δt time step. The difference is that here we do the half time step for all variables, and we thus have to evaluate F twice. The error for each time step of this method is also $O(\Delta t^3)$, and this method works for all ODEs, not just the ones that have a position-velocity separation.

Example

Let us again solve

$$f'(t) = \sqrt{f(t)}, \quad f(0) = 1 \tag{3.13}$$

with $\Delta t = 0.1$ but now using the second order Runge–Kutta method.

The first time step is done by evaluating

$$\begin{aligned} k_1 &= \sqrt{f(0)} = 1, \\ k_2 &= \sqrt{f(0) + \frac{1}{2} k_1 \Delta t} = \sqrt{1 + \frac{1}{2} \cdot 0.1} \approx 1.0247, \end{aligned}$$

and then

$$f(\Delta t) = f(0.1) \approx f(0) + k_2 \, \Delta t \approx 1.10247.$$

The exact value of $f(0.1)$ is 1.1025, showing that this method achieves much better results than what we obtained using the Euler method [Eq. (3.8)].

The most famous Runge–Kutta method is the simplest fourth order version:

Fourth Order Runge–Kutta (RK4)

For each time step evaluate

$$\begin{aligned}
k_1 &= F(f(t), t) \\
k_2 &= F(f(t) + \frac{1}{2}k_1\Delta t, t + \frac{1}{2}\Delta t) \\
k_3 &= F(f(t) + \frac{1}{2}k_2\Delta t, t + \frac{1}{2}\Delta t) \\
k_4 &= F(f(t) + k_3\Delta t, t + \Delta t) \\
f(t + \Delta t) &= f(t) + \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]\Delta t
\end{aligned} \tag{3.14}$$

The error of this method is $\mathcal{O}(\Delta t^5)$ per time step, which allows for significantly larger step sizes than that required by the Euler method.

Runge–Kutta methods are slightly harder to derive than finite difference coefficients. Their derivation, nonetheless, follow the exact same approach: compare with the general Taylor expansion to ensure the highest order of accuracy. In contrast to finite difference schemes, however, Runge–Kutta methods allow for some freedom of choice. To derive the RK4 scheme above, for instance, you begin by specifying that you want a scheme using the specified k_1, k_2, k_3, k_4 and then calculate the coefficients of these terms to determine $f(t+\Delta t)$ with best precision (i.e. you derive 1/6, 2/6, 2/6, 1/6). Different Runge–Kutta schemes can be obtained with difference choices of k's. In fact, the above scheme is not the best fourth order Runge–Kutta scheme, but it is the easiest one to implement.

Runge–Kutta schemes have one more trick up their sleeve: they provide a method to determine the best choice of Δt. Because Runge–Kutta methods of different order have different accuracy, we can do a time step with two different Runge–Kutta methods and estimate the error by their difference.

For instance, we know that for a fourth order Runge–Kutta method we have $f(t + \Delta t) = f(t) + \Delta f_{RK4} + \mathcal{O}(\Delta t^5)$, whereas a fifth order method would have $f(t+\Delta t) = f(t) + \Delta f_{RK5} + \mathcal{O}(\Delta t^6)$. Thus the difference $E = |\Delta f_{RK4} - \Delta f_{RK5}|$ will be of order $\mathcal{O}(\Delta t^5)$. Typically we want to choose the largest Δt while ensuring that the error remains smaller than some tolerance ϵ. Choosing Δt to be slightly smaller (say, a factor 0.9) than required, we find the perhaps most widely used solver for initial-value problems:

Adaptive Runge–Kutta 4(5) — RK45 / ode45

Choose a desired error tolerence ϵ.
Then for each time step

1. Compute Δf_{RK4} using a RK4 scheme
2. Compute Δf_{RK5} using a RK5 scheme
3. Estimate the error $E = |\Delta f_{RK4} - \Delta f_{RK5}|$
4. Choose a new $\Delta t = 0.9 \left(\frac{\epsilon}{E}\right)^{1/5} \Delta t_{\text{old}}$

5.
- If $E > \epsilon$ redo the time step with the new Δt
- Otherwise update $f(t + \Delta t_{\text{old}}) = f(t) + \Delta f_{RK5}$.

Using this scheme, large time steps will be taken when the solution is smooth and small time steps will be taken when the solution varies rapidly. This is computationally much more efficient than taking the same time step at all times.

The method is not fully specified in the above presentation, as there is still some choice in which Runge–Kutta formulas to use. A brilliant scheme, and the one most widely in use, is the Dormand–Prince method, which uses a fourth and fifth order scheme that share most of their k-expressions.

3.1.2 Implicit Time-Stepping

Runge–Kutta schemes allow for a large step size Δt while still ensuring good accuracy. Despite this, some differential equations still require so small Δt that it becomes impractical to solve them using the schemes presented thus far. Such problems are called *stiff*. This terminology does not have a precise definition, so we will just stick with this pragmatic version: *Stiff differential equations* are those that require impractically small Δt to be solved. For stiff problems, if Δt is not chosen small enough, the numerical solution will tend to become *unstable* and diverge to ± infinity.

Luckily, *implicit methods* solve much of the problem created by stiff equations. The downside is that these methods tend to be harder to implement than *explicit* methods (those just presented). Here we present just a few examples of implicit methods, but note that implicit methods become crucial once we have to solve partial differential equations in the following chapter.

In deriving the Euler Method we used the forward finite difference scheme. Had we instead used the backward scheme we would have obtained the Implicit Euler Method:

Implicit Euler Method

$$\boldsymbol{f}(t+\Delta t) = \boldsymbol{f}(t) + \boldsymbol{F}(\boldsymbol{f}(t+\Delta t), t+\Delta t)\,\Delta t \tag{3.15}$$

Note that the right-hand side now contains $f(t+\Delta t)$, which is what we are trying to calculate. This is the reason these methods are called implicit: the methods only provide implicit equations.

Example

Consider the initial-value problem

$$f'(t) = -af(t), \quad f(0) = f_0 > 0. \tag{3.16}$$

Using explicit Euler, the update rule would be

$$f(t+\Delta t) = f(t) - af(t)\Delta t. \tag{3.17}$$

Note that if $\Delta t > \frac{2}{a}$, this scheme becomes unstable as even though $f(t)$ should decay, it will instead do increasingly large

oscillations. In fact even for $\Delta t > \frac{1}{a}$ it will give solutions that have negative values, which should never be the case for the given equation.
In contrast the Implicit Euler Method gives us the update rule

$$f(t + \Delta t) = f(t) - af(t + \Delta t)\Delta t. \tag{3.18}$$

For this simple differential equation we can solve the implicit equation analytically. Thus we can rewrite the update rule to

$$f(t + \Delta t) = \frac{f(t)}{1 + a\Delta t}. \tag{3.19}$$

Clearly this implicit scheme will never have problems of oscillations or negative values, and in fact the method is stable for any positive value of Δt.

In the above example, Δt had to be chosen small for the explicit method to work. This could seem like not so big an issue, as $1/a$ sets the time scale of the problem, and thus it seems okay that we have to choose $\Delta t \ll 1/a$. We present one more example to illustrate why this can be a problem:

Example

Consider

$$\begin{aligned} f_1'(t) &= -a\left(f_1(t) + f_2(t)\right) - b\left(f_1(t) - f_2(t)\right), \\ f_2'(t) &= -a\left(f_1(t) + f_2(t)\right) + b\left(f_1(t) - f_2(t)\right). \end{aligned}$$

This problem has two time scales: $1/a$ and $1/b$. For initial conditions $f_1(0) = 1$ and $f_2(0) = 0$, the system has the solution

$$\begin{aligned} f_1(t) &= \frac{1}{2}\left(e^{-2at} + e^{-2bt}\right), \\ f_2(t) &= \frac{1}{2}\left(e^{-2at} - e^{-2bt}\right). \end{aligned}$$

Note that if, say, $a \ll b$ then e^{-2bt} will very quickly become negligible. Yet, for the explicit method to work, we will nonetheless be forced to choose $\Delta t \ll 1/b$. In other words: even in cases where the shortest time scale does nothing important for the final solution, we are still forced to use a small time scale in our time steps for explicit methods.
The implicit scheme for the above equation is

$$f_1(t + \Delta t) = \frac{(1 + a\Delta t + b\Delta t)\, f_1(t) + (b - a)\Delta t\, f_2(t)}{(1 + 2a\Delta t)(1 + 2b\Delta t)},$$

and similarly for $f_2(t + \Delta t)$.
For the implicit time step, a and b appear in both the numerator and denominator, which stabilises the scheme as, even for

large Δt, large values of e.g. b will not drive the right-hand side to be large.

The above example shows a typical situation, the classic example of which is the simulation of chemical reactions, where reaction rates often differ by many orders of magnitudes and often change as a function of time. In such cases, implicit methods become crucial.

In our two examples, we have solved the implicit equations analytically. This is not practical for large systems of equations. In these cases, the equations are solved numerically, and thus each time step will require a system of equations to be solved. For non-linear equations this can be quite cumbersome, and iterative methods should be used.

While the Implicit Euler Method is much more stable, its accuracy is similar to that of the Euler scheme, i.e. the error of each time step is of order $O(\Delta t^2)$. Implicit Runge–Kutta schemes also exist, and we will give just one example of such:

Crank–Nicolson Method

For each time step solve the following system of equations

$$\begin{aligned} k_1 &= F(f(t), t) \\ k_2 &= F(f(t+\Delta t), t+\Delta t) \\ f(t+\Delta t) &= f(t) + \frac{1}{2}(k_1 + k_2)\Delta t \end{aligned} \tag{3.20}$$

This implicit method has an error of order $O(\Delta t^3)$, but even though it is more stable the explicit Euler, it is not stable for any value of Δt.

3.2 Boundary-Value Problems

As we did for initial-value problems, we could present methods for boundary-value problems for a general ODE. However, we find that it is much easier to explain these methods if we work with a specific example. For this reason we will present methods that solve boundary-value problems that involve ODEs of the form

$$f''(x) = g(x), \quad x \in [a, b], \tag{3.21}$$

where $g(x)$ is some specified function. The methods presented should generalise quite easily to other linear problems, and we will end this chapter by discussing how to deal with non-linearities.

There are two main types of boundary conditions:

Dirichlet Boundary Condition

A Dirichlet boundary condition specifies the value of the function at the border, e.g.:

$$f(a) = \alpha \tag{3.22}$$

Neumann Boundary Condition

A Neumann boundary condition specifies the value of the derivative of a function at the border, e.g.:

$$f'(a) = \alpha \tag{3.23}$$

Naturally, combinations of such conditions could also be used as boundary conditions, as e.g. in Robin boundary conditions which specifies for instance $u_1 f(a) + u_2 f'(a) = \alpha$.

We will consider Eq. (3.21) with the boundary conditions

$$f(a) = \alpha, \qquad f'(b) = \beta. \tag{3.24}$$

3.2.1 Shooting Method

Consider that if we instead of the second boundary condition required $f'(a) = \beta$, then we would have an initial-value problem that we knew how to solve. This connection leads to a simple method called the Shooting Method: We guess enough boundary conditions at $x = a$ to be able to solve the ODE as an initial-value problem (shoot) and then readjust these guessed boundary conditions until the solution satisfies the boundary conditions required at $x = b$. For our example problem:

The Shooting Method

Find a γ which solves the equation

$$f'(b) = \beta, \tag{3.25}$$

where $f'(b)$ is evaluated from solving the initial-value problem of ODE with boundary condition

$$f(a) = \alpha, \qquad f'(a) = \gamma \tag{3.26}$$

Once γ is found, the initial-value problem solves the boundary-value problem.

This method requires some technique for solving numerically for γ, which can be as simple as trying γ and $\gamma + \epsilon$ and move γ in the direction which decreases the error as specified by $|f'(b) - \beta|$, or something more fancy (such as Powell's Method, which will not be discussed here).

The Shooting Method is easy to understand, but not trivial to implement well. It also does not generalise well to higher order equations, which have many boundary conditions, and in particular it is virtually useless in the context of partial differential equations, which will be the subject of the next chapter.

3.2.2 Finite Difference Method

In contrast to the shooting method, we will instead focus on methods that solve directly for the values of $f(x)$ at all values of x simultaneously. In the finite difference method, we discretise x, $f(x)$ and $g(x)$ and store the value of these as vectors

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}, \qquad \boldsymbol{f} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}, \qquad \boldsymbol{g} = \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_{N-1} \\ g_N \end{pmatrix}, \tag{3.27}$$

where $x_1 = a$ and $x_N = b$. Note that $\boldsymbol{f}$ is not known, as this is the one we are solving for. For regular grids we will have a constant spacing $x_{i+1} - x_i = \Delta x$ for all i, which we will assume for now.

Using the central finite difference scheme of Eq. (2.11), we have

$$f_i'' \approx \frac{f_{i+1} + f_{i-1} - 2f_i}{\Delta x^2} \tag{3.28}$$

for $2 \leq i \leq N - 1$. So except for the edge case $i = 1$ and $i = N$, we can write our discretised ODE as a matrix equation:

$$f'' = \frac{1}{\Delta x^2} \begin{pmatrix} ? & ? & ? & ? & \cdots & ? & ? & ? & ? \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ ? & ? & ? & ? & \cdots & ? & ? & ? & ? \end{pmatrix} \boldsymbol{f} \tag{3.29}$$

We could use the central scheme to fill out the first and last rows if we were solving a periodic problem. For the present problem we could also use a forward or backward finite difference scheme to fill out these rows to estimate the second derivative there. However, we do not need to do this, as we instead need to use these rows to enforce the boundary conditions. At $x = a$ we have to enforce $f_1 = \alpha$. At $x = b$ we need to enforce a Neumann boundary condition. We do this by using backward scheme for the first derivative. We could for instance enforce $f_N - f_{N-1} = \beta \Delta x$. However, as we use a second order scheme with error $\mathcal{O}(\Delta x^2)$, we should do the same for the boundary condition. The backward scheme of second order is given by $\frac{3}{2} f_N - 2 f_{N-1} + \frac{1}{2} f_{N-2} = \beta \Delta x$.

All in all, the finite difference version of Eq. (3.21) with boundary conditions as given by Eq. (3.24) is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{2\Delta x} & \frac{-2}{\Delta x} & \frac{3}{2\Delta x} \end{pmatrix} \boldsymbol{f} = \begin{pmatrix} \alpha \\ g_2 \\ g_3 \\ \vdots \\ g_{N-2} \\ g_{N-1} \\ \beta \end{pmatrix}. \tag{3.30}$$

This linear equation implements all requirements to the solution of our boundary-value problem: the first row implements the left boundary condition, the last row implements the right boundary condition, and the inner rows implement the differential equation that the solution must obey. It is a simple matter to solve the above equation on a computer by using a linear algebra library. This is the finite difference method.

Finite Difference Method

Use a finite difference scheme to rewrite the ODE as an approximate matrix equation,

$$\mathbb{A}\boldsymbol{f} = \boldsymbol{b}. \tag{3.31}$$

For each boundary condition, choose a suitable row and change those rows in $\mathbb{A}$ and $\boldsymbol{b}$ with finite difference approximations to the boundary conditions.

Finally solve

$$\mathbb{A}_{\mathrm{bc}}\boldsymbol{f} = \boldsymbol{b}_{\mathrm{bc}} \tag{3.32}$$

to find f, where $_{\mathrm{bc}}$ denotes the updated arrays.

Note that it is not necessary to invert $\mathbb{A}_{\mathrm{bc}}$, as faster methods exists that solve for $\boldsymbol{f}$ without doing explicit inversions. Also note that most entries of $\mathbb{A}_{\mathrm{bc}}$ will be zeros, it can there be useful to use sparse matrix representations. This will typically also give speed improvements.

This method is important, and will be used extensively also for partial differential equations. Therefore we present one more example on a slightly different ODE:

Example

Consider

$$f''(x) + h(x)f'(x) = 0, \quad x \in [a, b] \tag{3.33}$$

with the same boundary conditions (3.24). We can again use the matrix of Eq. (3.29) for the second derivative. Let us call this $\mathbb{A}_2$. In a similar fashion a first derivative matrix is

$$\mathbb{A}_1 = \frac{1}{2\Delta x}\begin{pmatrix} -3 & 4 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -4 & 3 \end{pmatrix},$$

where we used Eq. (2.9) for the inner rows and the forward and backward schemes for first end last rows. The total matrix representing the differential equation of Eq. (3.33) is therefore

$$\mathbb{A} = \mathbb{A}_2 + \operatorname{diag}(\boldsymbol{h})\,\mathbb{A}_1, \tag{3.34}$$

where $\operatorname{diag}(\boldsymbol{h})$ is a matrix of zeros with $\boldsymbol{h}$ filled along its diagonal, and matrix multiplication is implied. Now the boundary conditions can be applied to $\mathbb{A}$. The right-hand side is zero, so $\boldsymbol{b}_{\text{bc}} = (\alpha, 0, 0, \cdots, 0, 0, \beta)^T$.

Finally, note that if the equation had been

$$f''(x) + (h(x)f(x))' = 0, \quad x \in [a, b], \tag{3.35}$$

we would use[a]

$$\mathbb{A} = \mathbb{A}_2 + \mathbb{A}_1\operatorname{diag}(\boldsymbol{h}), \tag{3.36}$$

since order matters in matrix multiplication, and this specifies if we want to apply differentiation to h or not, since the matrices act from right to left on $\boldsymbol{f}$.

[a] The analog between the ODEs and the matrices becomes more clear if we write the ODE term with operator notation: $h(x)\partial_x f(x)$ and $\partial_x[h(x)f(x)]$. With this notation we simply replace $h(x)$ with $\mathrm{diag}(\boldsymbol{h})$ and ∂_x with $\mathbb{A}_1$ to obtain the finite difference matrix.

Note that in this example we could also have used $(h(x)f(x))' = h(x)f'(x) + h'(x)f(x)$.

Our examples have used a regular grid-spacing of Δx. The method, nevertheless, works for irregular grid-spacing as well. When constructing $\mathbb{A}$ you can then use e.g. Eq. (2.13) to derive the formulas for the derivatives. This can be very useful if the solution to the equation of interest varies faster in some regions than others.

Although fantastic library functions exist for solving linear equations, one can find oneself in a situation where a custom solver is needed. For this reason we end this section with a simple method for solving linear equations of the form $\mathbb{A}\boldsymbol{f} = \boldsymbol{b}$.

Under the assumption that a solution exists, we can reformulate the problem as

$$\min_{\boldsymbol{f}} |\mathbb{A}\boldsymbol{f} - \boldsymbol{b}|^2, \tag{3.37}$$

i.e. find an $\boldsymbol{f}$ that minimises the square distance between $\mathbb{A}\boldsymbol{f}$ and $\boldsymbol{b}$. If the minimum is zero, we have found the solution. Writing

$\mathcal{L} = |\mathbb{A}\boldsymbol{f} - \boldsymbol{b}|^2$, we have that the gradient of $\mathcal{L}$ with respect to $\boldsymbol{f}$ is

$$\nabla\mathcal{L} = 2\mathbb{A}^T(\mathbb{A}\boldsymbol{f} - \boldsymbol{b}). \tag{3.38}$$

Thus, if we have a guess for $\boldsymbol{f}$ we can make a better guess by changing $\boldsymbol{f}$ in the direction of $-\nabla\mathcal{L}$. If $\nabla\mathcal{L}$ is zero, we have found the minimum.

Gradient Descent for Linear Equations

To find a solution $\boldsymbol{f}$ for

$$\mathbb{A}\boldsymbol{f} = \boldsymbol{b}, \tag{3.39}$$

choose an initial guess $\boldsymbol{f}_0$ and a small step size ϵ.
Then until convergence, do

$$\boldsymbol{f}_n = \boldsymbol{f}_{n-1} - 2\mathbb{A}^T(\mathbb{A}\boldsymbol{f}_{n-1} - \boldsymbol{b})\,\epsilon. \tag{3.40}$$

The final $\boldsymbol{f}_n$ will be the solution if $|\mathbb{A}\boldsymbol{f}_n - \boldsymbol{b}|^2$ is zero. Otherwise it will be the “least square solution” of the equation. Choosing a good step size ϵ is important for the speed and convergence of the algorithm. Note that ϵ is allowed to be different in each step.

As should be the case, we did not need to explicitly invert $\mathbb{A}$ to solve the linear equation. Much better versions of this method exist such as the method of *Conjugate Gradients*, which you will probably be able to find a library for your language of choice. Conceptually these are similar to the above with smart choices for the step size. We will furthermore present an alternative method [Eq. (4.32)] in the chapter on partial differential equations.

3.2.3 Spectral Methods

We will exemplify the spectral method for ODEs on the system defined by

$$f''(x) + \eta f'(x) = g(x) \tag{3.41}$$

on a periodic domain $[a, b]$. This ODE only has a solution if $\int_a^b g(x)\,\mathrm{d}x = 0$.

Using the ideas of spectral methods presented in the previous chapter, we can rewrite the ODE using a Fourier transform

$$-k^2 \tilde{f}(k) + i\eta k \tilde{f}(k) = \tilde{g}(k), \tag{3.42}$$

where ˜ denotes a Fourier transform. From this we immediately have in the discrete case

$$\boldsymbol{f} = \mathrm{ifft}\left(\frac{\mathrm{fft}(\boldsymbol{g})}{-\boldsymbol{k}^2 + i\eta\boldsymbol{k}}\right), \tag{3.43}$$

where $\boldsymbol{k}$ is the vector of wave numbers of the Fourier transform. Here, $\boldsymbol{k}^2$ means element-wise square. We note that the first wave number $\boldsymbol{k}_0$ is zero, and so we in principle we are dividing by zero for this term. But since $\int_0^b g(x)\,\mathrm{d}x = 0$, the corresponding Fourier component will also be zero. This “zero divided by zero” may be replaced by an actual zero to obtain the correct solution.

In general:

Spectral Method

1. Use fast fourier transforms and $\widetilde{f^{(n)}} = (ik)^n \tilde{f}$ to rewrite the equation in terms of k and $\tilde{f}$.
2. Solve the equation for $\tilde{f}$ taking special care for zero denominators.
3. Use the fast inverse fourier transform to obtain the solution.

The spectral method is quite simple to implement, but we will nonetheless give a quick numerical example, since there are small details that can easily be overlooked:

Example

Consider Eq. (3.41) on the domain $[0, 2\pi)$ with $\eta = 1$. We will solve this on the grid

$$\boldsymbol{x} = (0.00,\ 0.79,\ 1.57,\ 2.36,\ 3.14,\ 3.93,\ 4.71, 5.50),$$

i.e. with $\Delta x = \pi/8 \approx 0.79$. We will solve the ODE for a right-hand side function g that on our grid takes the values

$$\boldsymbol{g} = g(\boldsymbol{x}) = (1.0,\ 1.7,\ 0.0,\ -1.7,\ -1.0,\ 0.3,\ 0.0,\ -0.3).$$

Using the fast fourier transform we find[a]

$$\text{rfft}(\boldsymbol{g}) \approx (0,\ 4.0,\ -4.0\,i,\ 0,\ 0),$$

corresponding to wave numbers[b]

$$\boldsymbol{k} = (0.0,\ 1.0,\ 2.0,\ 3.0,\ 4.0).$$

Thus we have

$$\frac{\mathrm{rfft}(\boldsymbol{g})}{-\boldsymbol{k}^2 + i\boldsymbol{k}} = ({}^{0.0}/{0.0},\ -2 - 2\,i,\ -0.4 + 0.8\,i,\ 0,\ 0).$$

We replace ${}^{0.0}/{0.0}$ by zero, and then solve the PDE by calculating

$$\begin{aligned}\boldsymbol{f} &= \mathrm{ifft}\left(\frac{\mathrm{rfft}(\boldsymbol{g})}{-\boldsymbol{k}^2 + i\boldsymbol{k}}\right)\\ &= (-0.6,\ -0.2,\ 0.6,\ 0.9,\ 0.4,\ -0.2,\ -0.4,\ -0.5).\end{aligned}$$

In fact a constant can also be added to the above solution and will still solve the equation. This is because the ODE does not contain the value $f(x)$ but only its derivatives.

[a] We use the real fast fourier transform "rfft".

[b] These are simply multiplies of $1/\Delta x$. Your fft library will most likely have a utility function to calculate the frequencies, e.g. called "rfftfreq".

In the above example, we did not tell you what the function g was, but only defined it in terms of its values at the grid points. As can be deduced from the Fourier coefficients, our choice was

$$g(x) = \cos(x) + \sin(2x). \tag{3.44}$$

However, with only 8 points as used in the example, there is no way to tell the difference between this function and the function

$$g_2(x) = \cos(9x) + \sin(18\,x). \tag{3.45}$$

This means that we would have obtained the exact same solution if we had used g_2 and the same number of grid points. This effect is called *aliasing*. When using spectral methods there is therefore a simple rule that must be followed: use a grid-spacing Δx that is small enough to capture the highest frequencies in all functions.

Beyond being very easy to implement spectral methods also have a very high accuracy. For an n'th order finite difference method, the accuracy is $\mathcal{O}(\Delta x^n)$, which in terms of the number of grid point m is $\mathcal{O}(m^{-n})$, since $\Delta x \sim \frac{1}{m}$. The error for spectral methods is of order $\mathcal{O}(e^{-m})$. In other words, finite difference methods are polynomial in their error, whereas spectral methods are exponential. So: if you can apply them, do apply them. We note, however, that spectral methods require the solution to be smooth. Small discontinuities, e.g. in a boundary condition, can completely ruin the method.

As mentioned in the previous chapter, spectral methods also exist for non-periodic domains, but the formulas differ, as different basis functions need to be used.

3.2.4 Non-linear Problems

Non-linear terms in ODEs do not pose much of a problem for initial-value problems. However, for boundary-value problems they prevent the problem from being formulated as a linear system of equations. The shooting method can still be used unaltered in these cases. However, the shooting method does not generalise well to PDEs, and we will therefore focus on other methods. Here we present a simple alternative that can be used with both the finite difference method and with spectral methods.

Consider a non-linear ODE such as

$$\mathcal{D}f(x) = N(f(x), x), \tag{3.46}$$

where $\mathcal{D}$ is some linear differential operator and N is a non-linear function of f. Using finite difference, we know to discretise the left-hand side. So we have an equation of the form

$$\mathbb{A}\boldsymbol{f} = N(\boldsymbol{f}). \tag{3.47}$$

If you have N grid points, this is simply a system of N non-linear equations. There are many algorithms to solve such systems of equations directly, and using these could be a good approach.

A simple alternative is to use the relaxation method:

The Relaxation Method

To solve for $\boldsymbol{f}$ in a system of non-linear equations, rewrite the equations in form

$$f_i = F(\{f_j\}), \tag{3.48}$$

where the right-hand side also is allowed to depend on f_i. Then choose a starting guess $\boldsymbol{f}$ and until convergence for all i not on the boundary do

$$f_i \leftarrow F(\{f_j\}). \tag{3.49}$$

The right-hand side is evaluated for all i before being reassigned to update f_i. Update boundary values using boundary conditions after each iteration.

This method can be used in many ways, and how well it works can vary significantly depending on the problem at hand.

Example

To solve Eq. (3.47), choose a starting guess $\boldsymbol{f}$ and repeat

$$\boldsymbol{f} \leftarrow \mathbb{A}^{-1} N(\boldsymbol{f}). \tag{3.50}$$

When the method works, the solution will be found after a number of iterations. There is no guarantee that this method convergences though, so it should be checked that the solution found solves the equation. Also note that in principle, many solutions could exist. Which solution is found depends on the initial guess.

For a concrete problem, we could also use the relaxation method in the following way

Example

Consider

$$f''(x) = N(f(x), x). \tag{3.51}$$

Using central differences we can discretise this as

$$\frac{f_{i-1} + f_{i+1} - 2f_i}{\Delta x^2} = N(f_i, x) \tag{3.52}$$

We can rewrite this in form of Eq. (3.48) as

$$f_i = \frac{1}{2}\left(f_{i-1} + f_{i+1} - N(f_i, x_i)\Delta x^2\right), \tag{3.53}$$

which can be iterated to find the solution. In this case we did not even have to form a matrix.

Note that this method is also a viable approach to solve linear equations. We further note that convergence of the method depends on the initial choice and is not guaranteed.

A separate approach is to use root-finding algorithms to solve the non-linear problem. The non-linear equation resulting from an ODE can always be formulated as an n-dimensional equation of the form

$$\mathbf{F}(\boldsymbol{x}) = 0. \tag{3.54}$$

To find an $\boldsymbol{x}$ that solves the above non-linear equation, one could use

Newton's Method

To solve for $\boldsymbol{x}$ in a system of non-linear equations, rewrite the equation in the form of (3.54).
From an initial guess x_0, use the Jacobian

$$J_F = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix} \tag{3.55}$$

and iteratively solve the linear system of equations

$$J_F(\boldsymbol{x}_n)\,\boldsymbol{a} = -\mathbf{F}(\boldsymbol{x}_n) \tag{3.56}$$

for $\boldsymbol{a}$ and set

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \boldsymbol{a}. \tag{3.57}$$

Iterate until convergence.

The method requires that the Jacobian may be calculated. It is naturally most efficient when this can be calculated analytically, but also works with numerical approximations of the Jacobian.

Example

Consider the equations

$$\begin{aligned} y^3 &= 1 - x^2, \\ y &= -e^x. \end{aligned} \tag{3.58}$$

Rewriting we have

$$\mathbf{F}(x, y) = \begin{pmatrix} y^3 + x^2 - 1 \\ y + e^x \end{pmatrix}. \tag{3.59}$$

The Jacobian is

$$J_F(x, y) = \begin{pmatrix} 2x & 3y^2 \\ e^x & 1 \end{pmatrix}. \tag{3.60}$$

In this simple case, we can simply invert the Jacobian to find

$$J_F^{-1}(x, y) = \frac{1}{e^x/y^2 + 2x} \begin{pmatrix} 1 & 1/y^2 \\ -e^x & 2x \end{pmatrix}. \tag{3.61}$$

Choosing e.g. $\boldsymbol{x}_0 = (1, 1)^T$ we iterate

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - J_F^{-1}(\boldsymbol{x}_n)\,\mathbf{F}(\boldsymbol{x}_n). \tag{3.62}$$

In this simple case we can actually write out analytically the full update

$$\begin{aligned} x &\leftarrow x - x^2 + y^2\left(3e^x + 2y\right), \\ y &\leftarrow y + e^x\left(\left(x-2\right)x + y^3\right) - 2xy \end{aligned} \tag{3.63}$$

After just seven iterations we find the to a good accuracy the correct solution $x = -1.02297$ and $y = -0.359525$.

Partial Differential Equations

Most of the principles that we have discussed for Ordinary Differential Equations (ODEs) also apply for Partial Differential Equations (PDEs). Nonetheless, solving PDEs is generally harder than solving ODEs. This is true for analytical approaches and numerical. Implementation-wise, PDEs are harder to write solvers for, since we have to think about the grids on which the PDEs should be solved — for ODEs, all we had to think about was a one-dimensional x-axis. PDEs are also harder for the computer, since typically a d-dimensional problem will have N^d variables to be solved for if we want N grid points in each direction. In 3D this very quickly becomes unmanageable.

We will consider two types of PDE problem formulations: time-dependent problems that is a mix of initial-value and boundary-value problems, and time-independent problems that are full boundary-value problems. The former will in many cases be the easier one to approach. In particular, we will exemplify the methods using the diffusion (heat) equation

$$\frac{\partial f(t,\boldsymbol{x})}{\partial t} = \gamma \nabla^2 f(t,\boldsymbol{x}), \tag{4.1}$$

the advection equation

$$\frac{\partial f(t,\boldsymbol{x})}{\partial t} = \nabla \cdot (\boldsymbol{u} f(t,\boldsymbol{x})), \tag{4.2}$$

and Poisson's equation

$$\nabla^2 f(\boldsymbol{x}) = g(\boldsymbol{x}). \tag{4.3}$$

Nevertheless, the methods we develop will have general applicability. We will write the general time-dependent PDE as

$$\frac{\partial f(t,\boldsymbol{x})}{\partial t} = \mathcal{D} f(t,\boldsymbol{x}) + g(\boldsymbol{x},t), \tag{4.4}$$

and the general time-independent PDE as

$$\mathcal{D} f(t,\boldsymbol{x}) = g(\boldsymbol{x}), \tag{4.5}$$

where $\mathcal{D}$ is some differential operator. Time-dependent problems need both initial-value and boundary-value conditions to be fully specified. We postpone a discussion of non-linear PDEs for the end of the chapter.

For ODEs, we discovered that some methods were more stable than others. This is a critical subject for PDEs as many PDEs simply cannot be solved with some methods. We begin this chapter by introducing a simple method for solving some time-dependent PDEs and then show when this approach fails. The following sections will then be dedicated to more stable (implicit) methods.

4.1 Explicit Methods & Stability Analysis

Probably the simplest scheme for PDEs follows by combining central schemes for calculating derivatives with the Euler Method in the most straightforward way. The method is, for obvious reasons, known as the Forward-Time-Central-Space (FTCS) scheme:

FTCS Scheme

To solve PDEs of the form Eq. (4.4), use forward Euler to discretise the time-derivative, and a central scheme to discretise spatial derivatives.

For one spatial dimension this becomes:

$$\begin{aligned} f(t+\Delta t, x_i) = f(t, x_i) + D(\{f(t,\, x_j)\})\Delta t \\ + g(t, x_i)\Delta t \end{aligned} \tag{4.6}$$

where D is the discretised version of $\mathcal{D}$.

After each time step, ensure that boundary conditions are met by updating the values of f at the borders.

Note that this method can only be used for time-dependent problems. It cannot be used to tackle equations of the form of Eq. (4.5).

Example

Consider

$$\frac{\partial f(t, \boldsymbol{x})}{\partial t} = \gamma \frac{\partial^2 f(t, \boldsymbol{x})}{\partial x^2} \tag{4.7}$$

with boundary conditions $f(t,a) = \alpha$ and $f(t,b) = \beta$. The FTCS Scheme for this problems is to do

$$f(t+\Delta t, x_i) = f(t,x_i)+\gamma \frac{f(t,x_{i-1}) + f(t,x_{i+1}) - 2f(t,x_i)}{\Delta x^2}\Delta t \tag{4.8}$$

for all grid points x_i except for $x_0 = a$ and $x_N = b$, as these are set by the boundary conditions.

We will now discuss the stability of this scheme on the example problem above. That is, we will discuss how small Δt has to be chosen in order for the method to be stable. It is harder to do this type of analysis for PDEs than for ODEs. The approach we will take is called *Von Neumann stability analysis*. The analysis considers the growth-rate of the Fourier modes of the numerical solution, and the idea is to require that no mode may grow infinitely big.

For simplicity we consider periodic boundary conditions, although this will not make a difference for the stability condition that we derive. In this case we can write the solution as a Fourier series. Using N grid points we have[1]

$$f(t,x) = \sum_{n=-N/2}^{N/2} c_n(t)\, e^{ik_n x}, \tag{4.9}$$

where $k_n = \pi\, n\, \Delta x$. Using this equation in Eq. (4.8) and Fourier

[1] Exact expression depends on whether N is odd or even.

orthogonality, we obtain

$$c_n(t+\Delta t) = c_n(t) + \gamma \frac{e^{-ik_n\Delta x}c_n(t) + e^{ik_n\Delta x}c_n(t) - 2c_n(t)}{\Delta x^2}\Delta t, \tag{4.10}$$

which can be rewritten as

$$\frac{c_n(t+\Delta t)}{c_n(t)} = 1 + \left(e^{-ik_n\Delta x} + e^{ik_n\Delta x} - 2\right)\frac{\gamma\Delta t}{\Delta x^2}. \tag{4.11}$$

Note that the right-hand side does not depend on t. This equation tells you what to multiply $c_n(t)$ by to get $c_n(t+\Delta t)$. If for any n

$$\left|1 + \left(e^{-ik_n\Delta x} + e^{ik_n\Delta x} - 2\right)\frac{\gamma\Delta t}{\Delta x^2}\right| > 1, \tag{4.12}$$

then $c_n \longrightarrow \infty$ as we run our numerical simulation. This means our scheme is unstable, and some mode will grow indefinitely big.[2] Requiring Eq. (4.12) to be false for all n leads to

$$\Delta t \leq \frac{\Delta x^2}{2\gamma}. \tag{4.13}$$

This is the Von Neumann stability criteria for the FTCS Scheme on the diffusion equation. There are two main takeaways: (1) The better spatial resolution you require, the smaller Δt you have to choose, and (2) the larger the diffusion constant γ, the more stable the scheme is. Finally, we note that stability is not the same as

[2] Note that c_0 does not change with time. This is because the steady state of the diffusion equation is a constant).

accuracy. You can have a stable scheme that is very inaccurate. Stability just means that the numerical solution does not blow up.

Now consider a one-dimensional advection equation

$$\frac{\partial f(t,x)}{\partial t} = u\frac{\partial f(t,x)}{\partial x}, \tag{4.14}$$

which has the FTCS Scheme

$$f(t+\Delta t, x_i) = f(t,x_i) + u\,\frac{f(t,x_{i+1}) - (t,x_{i-1})}{2\Delta x}\Delta t. \tag{4.15}$$

Using the Von Neumann approach, we find in this case

$$\frac{c_n(t+\Delta t)}{c_n(t)} = 1 + \left(e^{ik_n\Delta x} - e^{-ik_n\Delta x}\right)\frac{u\Delta t}{2\Delta x}. \tag{4.16}$$

The condition for this equation is therefore[3]

$$\left|1 + i\sin(k_n\Delta x)\frac{u\Delta t}{4\Delta x}\right| \leq 1. \tag{4.17}$$

But we can never satisfy this condition for all n. Therefore the FCTS scheme is unstable for *any* value of Δt for the advection equation.

For this reason we will do not recommend the FCTS scheme unless you are certain that you are applying it to a system for which you know it is stable. Explicit schemes that work for the advection equation do exist: for instance the *Lax–Wendroff method*, which

[3] Using $e^{ix} + e^{-ix} = 2i\sin(x)$.

explicitly expands the time-derivative to second order, or *upwind schemes*, which replace the central finite difference scheme with forward or backward schemes depending on the sign of u. We will not discuss these methods here, but instead focus on implicit schemes which have more broad applicability.

4.2 Finite Difference Method

If you have not yet read and understood Sec 3.1.2 and Sec. 3.2.2, it is recommended that you do so before continuing here. The methods presented here are very similar, but slightly more involved than those discussed for ODEs.

4.2.1 Time-dependent Problems

Using implicit time stepping and our knowledge of solving boundary-value problems for ODEs immediatetely gives us a very stable scheme for time-dependent PDEs with one spatial dimension. For Eq. (4.4), implicit time stepping yields[4]

$$\begin{aligned} f(t+\Delta t, x_i) = f(t, x_i) &+ D(\{f(t+\Delta t,\, x_j)\})\,\Delta t \\ &+ g(t, x_i)\,\Delta t, \end{aligned} \tag{4.18}$$

where D is some discretised version of $\mathcal{D}$. To use finite differences we write f, just as we did for ODEs, as a vector of its values: $\boldsymbol{f}$.

[4] It is matter of choice whether it should be $g(t, x_i)$ or $g(t+\Delta t, x_i)$.

This allows us recast the equation as a vector equation:

$$\boldsymbol{f}(t+\Delta t) = \boldsymbol{f}(t) + \mathbb{D}\boldsymbol{f}(t+\Delta t)\Delta t + \boldsymbol{g}(t)\Delta t. \tag{4.19}$$

Which can also be written as

$$(\mathbb{I} - \mathbb{D}\Delta t)\,\boldsymbol{f}(t+\Delta t) = \boldsymbol{f}(t) + \boldsymbol{g}(t)\Delta t, \tag{4.20}$$

where $\mathbb{I}$ is the identity matrix. Thus, in the notation of Eq. (3.31) that we used for boundary-value problems, we have

$$\mathbb{A} = \mathbb{I} - \mathbb{D}\Delta t, \tag{4.21}$$

and

$$\boldsymbol{b} = \boldsymbol{f}(t) + \boldsymbol{g}(t)\Delta t. \tag{4.22}$$

Now we can apply boundary-conditions to $\mathbb{A}$ and $\boldsymbol{b}$ and then solve the boundary-value problem for each step time.

This approach also generalises to higher dimensions.

Implicit Finite Difference for Time-Dependent PDEs

1. Use an implicit time stepping scheme such as backward Euler.

2. Discretise the spatial derivatives using a finite difference scheme.

3. Each time step in the simulation is then taken by solving the resulting boundary-value problem.

Example

Consider the diffusion equation with a source term

$$\frac{\partial f(t,\boldsymbol{x})}{\partial t} = \gamma \frac{\partial^2 f(t,\boldsymbol{x})}{\partial x^2} + g(x) \tag{4.23}$$

on $[a,b]$ with boundary conditions $f(t,a) = \alpha$ and $f(t,b) = \beta$. Using Eq. (3.29) and Eq. (4.21), we have

$$\mathbb{A} = \mathbb{I} - \frac{\Delta t}{\Delta x^2} \begin{pmatrix} ? & ? & ? & ? & \cdots & ? & ? & ? & ? \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ ? & ? & ? & ? & \cdots & ? & ? & ? & ? \end{pmatrix}.$$

and

$$\boldsymbol{b} = \boldsymbol{f}(t) + \boldsymbol{g}\Delta t \tag{4.24}$$

Applying our boundary conditions we find

$$\mathbb{A}_{\text{bc}} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \frac{-\Delta t}{\Delta x^2} & 1 + \frac{2\Delta t}{\Delta x^2} & \frac{-\Delta t}{\Delta x^2} & 0 & \cdots & 0 & 0 \\ 0 & \frac{-\Delta t}{\Delta x^2} & 1 + \frac{2\Delta t}{\Delta x^2} & \frac{-\Delta t}{\Delta x^2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{-\Delta t}{\Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 + \frac{2\Delta t}{\Delta x^2} & \frac{-\Delta t}{\Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

and

$$\boldsymbol{b}_{\mathrm{bc}} = \begin{pmatrix} \alpha \\ f(t, x_2) + g(x_2)\Delta t \\ f(t, x_3) + g(x_3)\Delta t \\ \vdots \\ f(t, x_{N-2}) + g(x_{N-2})\Delta t \\ f(t, x_{N-1}) + g(x_{N-1})\Delta t \\ \beta \end{pmatrix}.$$

Each time step is then taken by solving for $\boldsymbol{f}(t + \Delta t)$ in

$$\mathbb{A}_{\mathrm{bc}}\boldsymbol{f}(t + \Delta t) = \boldsymbol{b}_{\mathrm{bc}} \tag{4.25}$$

Note that it makes sense to use an iterative solver (such as Eq. (3.40) or the method of conjugate gradients) to solve the linear equation, since we have an excellent initial guess for $\boldsymbol{f}(t + \Delta t)$ in the form of $\boldsymbol{f}(t)$.

If the problem is higher-dimensional, the approach is the same, except that the boundary-value problem to be solved for each time step is higher-dimensional. The next section is devoted to the solution of such problems.

4.2.2 Time-independent Problems

For problems such as Eq. (4.5) of dimensions larger than one, we need to think about how we represent the discretised version of

$f(\boldsymbol{x})$. We still intend to write the equation in the form

$$\mathbb{A}\boldsymbol{f} = \boldsymbol{b}, \tag{4.26}$$

where $\boldsymbol{f}$ is a vector. In the case of a two-dimensional problem, where we discretise both x and y with N grid points, $\boldsymbol{f}$ must contain N^2 values, one value for each location $\{x_i, y_j\}$. Likewise $\mathbb{A}$ will be an $N^2 \times N^2$ matrix.

It is a matter of choice how to order the values[5], but to make things simple it makes sense to order them in the vector as

$$\boldsymbol{f} = \begin{pmatrix} f(x_1, y_1) \\ f(x_2, y_1) \\ \vdots \\ f(x_{N-1}, y_1) \\ f(x_N, y_1) \\ f(x_1, y_2) \\ f(x_2, y_2) \end{pmatrix} \begin{pmatrix} f(x_3, y_2) \\ f(x_4, y_2) \\ \vdots \\ f(x_N, y_2) \\ f(x_1, y_3) \\ f(x_2, y_3) \\ \vdots \end{pmatrix} \vdots \begin{pmatrix} \vdots \\ f(x_N, y_{N-1}) \\ f(x_1, y_N) \\ f(x_2, y_N) \\ \vdots \\ f(x_{N-1}, y_N) \\ f(x_N, y_N) \end{pmatrix},$$

i.e. stacking $f(\boldsymbol{x}, y_1), f(\boldsymbol{x}, y_2), \cdots, f(\boldsymbol{x}, y_N)$. The same idea holds for higher dimensions.

[5] We could also formulate it as a tensor equation, which would avoid the issue of choosing an ordering. Most libraries will also provide a ‘tensorsolve’ function. We take the above approach since it is most standard, it is in line with what is taught in standard linear algebra courses, allows the use of all linear algebra functionalities provided by libraries. However, for simple cases the tensorial approach can give code that is easier to read/maintain.

For instance, consider the problem of representing $\partial_x f(x, y)$ as a finite difference equation. Let us do this explicitly for a periodic system with just $N = 4$ grid point.[6] We are then trying to solve for $\boldsymbol{f}$ which contains elements that represent the function as

$$\begin{tikzcd}
f(x_1, y_1) \arrow[r, dash] \arrow[d, dash] & f(x_2, y_1) \arrow[r, dash] \arrow[d, dash] & f(x_3, y_1) \arrow[r, dash] \arrow[d, dash] & f(x_4, y_1) \arrow[r, dash, dashed] \arrow[d, dash] & {} \\
f(x_1, y_2) \arrow[r, dash] \arrow[d, dash] & f(x_2, y_2) \arrow[r, dash] \arrow[d, dash] & f(x_3, y_2) \arrow[r, dash] \arrow[d, dash] & f(x_4, y_2) \arrow[r, dash, dashed] \arrow[d, dash] & {} \\
f(x_1, y_3) \arrow[r, dash] \arrow[d, dash] & f(x_2, y_3) \arrow[r, dash] \arrow[d, dash] & f(x_3, y_3) \arrow[r, dash] \arrow[d, dash] & f(x_4, y_3) \arrow[r, dash, dashed] \arrow[d, dash] & {} \\
f(x_1, y_4) \arrow[r, dash] \arrow[d, dash, dashed] & f(x_2, y_4) \arrow[r, dash] \arrow[d, dash, dashed] & f(x_3, y_4) \arrow[r, dash] \arrow[d, dash, dashed] & f(x_4, y_4) \arrow[r, dash, dashed] \arrow[d, dash, dashed] & {} \\
{} & {} & {} & {} &
\end{tikzcd}$$

[6] For finite differences, periodicity implies that e.g. x_1 and x_N are neighbouring points.

We use the central derivative scheme [Eq. (2.9)] and find $\partial_x f$ to be approximated by

$$\frac{1}{2\Delta x}\begin{pmatrix}
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0
\end{pmatrix}\begin{pmatrix}
f(x_1,y_1)\\ f(x_2,y_1)\\ f(x_3,y_1)\\ f(x_4,y_1)\\
f(x_1,y_2)\\ f(x_2,y_2)\\ f(x_3,y_2)\\ f(x_4,y_2)\\
f(x_1,y_3)\\ f(x_2,y_3)\\ f(x_3,y_3)\\ f(x_4,y_3)\\
f(x_1,y_4)\\ f(x_2,y_4)\\ f(x_3,y_4)\\ f(x_4,y_4)
\end{pmatrix}.$$

This is somewhat hard to read, so do not feel bad if you skip it: this is easier to program than to read! But do focus on single, random row and make sure to understand that one. For instance, the second row states $\partial_x f(x_2, y_1) = \frac{1}{2\Delta x}(f(x_3, y_1) - f(x_1, y_1))$. Note that the matrix would be very different if we needed $\partial_y f$. Also note that as we increase the number of grid points N, the fraction of zeros increases. It is thus a very good idea to use a sparse matrix representation.

For higher-dimensional PDEs we will often encounter the Laplacian: ∇^2. It is useful to a have a finite difference scheme handy specifically for this operator. The standard choice is to use central finite differences along all dimensions:

Discrete Laplace Operator

With regular grid-spacing the discrete Laplace operator is defined as

$$\begin{aligned}\nabla^2 f(x_i, y_j) &= (\partial_x^2 + \partial_y^2) f(x_i, y_j) \qquad (4.27)\\ &\approx \frac{1}{\Delta x^2}\big[f(x_{i-1}, y_j) + f(x_{i+1}, y_j)\\ &\qquad + f(x_i, y_{j-1}) + f(x_i, y_{j+1}) - 4f(x_i, y_j)\big]\end{aligned}$$

in two dimensions.
It generalises straighforwardly to higher dimensions.

For example, $\nabla^2 f(x, y)$ for our periodic $N = 4$ system is represented by the finite difference expression $\mathbb{L}\boldsymbol{f}$, where

$$\mathbb{L} = \frac{1}{\Delta x^2}\begin{pmatrix}
-4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\
1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1\\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1\\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4
\end{pmatrix}. \qquad (4.28)$$

Again, make sure you understand just one or two rows. That should be enough to understand the idea and enable you to write a program that generates such a matrix for any N.

Example

Consider

$$\nabla^2 f(x, y) = g(x, y). \tag{4.29}$$

The finite difference equation is then simply

$$\mathbb{L}\boldsymbol{f} = \boldsymbol{g}, \tag{4.30}$$

which can be solved using any linear solver after applying boundary conditions.

Constructing the matrix $\mathbb{A}$ allows us to use any library for solving the linear equation. This is typically very efficient. However, it does take some code to build $\mathbb{A}$, which sometimes you want to avoid. In this case we can turn to relaxation methods [Eq. (3.48)]. For the systems of equations that result from linear PDEs there is a specific version of the relaxation method that is easy to implement:[7]

Jacobi Method

A linear equation

$$\mathbb{A}\boldsymbol{f} = \boldsymbol{b} \tag{4.31}$$

[7] However, ease of implementation is at the of cost speed of convergence: the Jacobi method typically requires many iterations.

can be solved by splitting the matrix into its diagonal and off-diagonal parts $\mathbb{A} = \mathbb{D} + \mathbb{O}$. Here $\mathbb{D}$ only contains elements on the diagonal and $\mathbb{O}$ only contains elements off the diagonal. Then until convergence do

$$\boldsymbol{f} \leftarrow \mathbb{D}^{-1} \left(\boldsymbol{b} - \mathbb{O}\boldsymbol{f} \right) \tag{4.32}$$

Note that $\mathbb{D}^{-1}$ is trivial to calculate.

The method works if it holds for all rows i that $|\mathbb{A}_{ii}| \geq \sum_{j \neq i} |\mathbb{A}_{ij}|$, i.e. that for each row in the matrix, the diagonal element is at least as large as the sum of the rest of the elements in that row. Further, the inequality has to be strict $|\mathbb{A}_{ii}| > \sum_{j \neq i} |\mathbb{A}_{ij}|$ in at least one row (meaning the matrix is so-called *irreducibly diagonally dominant*).

The Jacobi method is a relaxation method, since it is just a simple rewrite of $\mathbb{A}\boldsymbol{f} = (\mathbb{D} + \mathbb{O})\boldsymbol{f} = \boldsymbol{b}$. A nice feature is that we have a condition for convergence (although in fact the method sometimes works even when the condition is not satisfied[8]). To get a feeling for the usefulness of this method, we turn to an example:

[8] The condition we have stated is *sufficient* for convergence, but not *necessary*. The *sufficient and necessary* condition is that all eigenvalues of $\mathbb{D}^{-1}\mathbb{O}$ are less than or equal to 1, and at least one eigenvalue is strictly less than 1.

Example

Consider again

$$\nabla^2 f(x, y) = g(x, y), \tag{4.33}$$

which we will again approximate using Eq. (4.27). For the Laplacian matrix $\mathbb{L}$ the condition to use the Jacobi method holds with equality: $|-4| \geq |1| + |1| + |1| + |1|$. A Dirichlet boundary condition [Eq. (3.22)] will make the full condition to use the Jacobi method true (in fact this method will typically also converge for periodic systems if the initial guess is not too terrible).

The method then requires us to iterate

$$\begin{aligned} f(x_i, y_j) \leftarrow \frac{1}{4}\big(&f(x_{i+1}, y_j) + f(x_{i-1}, y_j) \\ &+ f(x_i, y_{j+1}) + f(x_i, y_{j-1}) - g(x_i, y_j)\,\Delta x^2\big) \end{aligned} \tag{4.34}$$

where the right-hand side is evaluated for all i, j before the update. The updates at the boundary points depend on the boundary conditions.

Note that for this method $f(x, y)$ can be stored as a matrix instead of as a vector, since we do not need to put it in a shape suitable for library functions. This can simplify implementation quite a lot.

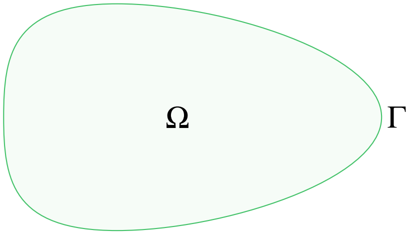
Many methods are intuitive to read. This one requires a bit more thought, so make sure you understand how to obtain Eq. (4.34) from Eq. (4.32). This method is simpler to implement, but not as efficient as those found in many modern linear algebra libraries.

4.3 Boundary Conditions

The boundary conditions for ODEs are very easy to specify, as these will just be at either end of a domain such as $[a, b]$. In higher dimensions, the boundary will not simply be points, but lines for 2D PDEs, faces for 3D PDEs, and so on.

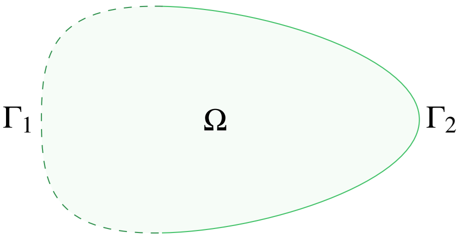
You will often find the notation used in which the region on which we solve is denoted as Ω, and the boundary Γ.[9] For an ODE, for example, we could solve on the domain given by the interval $\Omega = [a, b]$, while the boundary is the edge points $\Gamma = \{a, b\}$.

In two dimensions, we could illustrate it like this:



To specify different boundary conditions on different parts of the boundary, it is often divided into sections:

[9] It is also standard to use $\partial\Omega$ instead of Γ.



For instance, a boundary-value problem could be specified as

$$\begin{cases} \nabla^2 f(\boldsymbol{x}) = 0 & \boldsymbol{x} \in \Omega, \\ f(\boldsymbol{x}) = \alpha & \boldsymbol{x} \in \Gamma_1, \\ f(\boldsymbol{x}) = \beta & \boldsymbol{x} \in \Gamma_2. \end{cases} \tag{4.35}$$

Numerically, this simply means that we have to keep track of which of our discretised points belong to which part of the boundary.

As clear from the example above, Dirichlet boundary conditions are similar for PDEs as they are for ODEs. Neumann boundary conditions are generalised as

Neumann Boundary Conditions (PDEs)

A Neumann boundary condition specifies the value of the directional derivative along the boundary normal $\hat{n}$ of the function at the border, e.g.:

$$\left.\frac{\partial f}{\partial \hat{n}}\right|_{\boldsymbol{x}=\boldsymbol{a}} = (\nabla f \cdot \hat{n})\big|_{\boldsymbol{x}=\boldsymbol{a}} = \alpha \tag{4.36}$$

Example

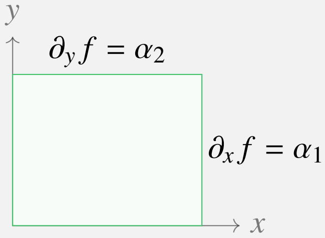
Consider

$$\nabla^2 f(x, y) = 0 \tag{4.37}$$

on a square $x \in [a_x, b_x]$ and $y \in [a_y, b_y]$. Suitable boundary conditions could be

$$\begin{cases} f(x, a_y) = f(a_x, y) = \beta \\ \partial_x f(b_x, y) = \alpha_1 \\ \partial_y f(x, b_y) = \alpha_2 \end{cases} \tag{4.38}$$

Here we used two Neumann conditions:



To implement boundary conditions numerically, we recommend simply changing rows in the matrix $\mathbb{A}$ to make $\mathbb{A}_{\text{bc}}$, as we have done so far in all examples. The rows that need to be changed are those that correspond to points on the border. If a border point has two boundary conditions, then two points need to be chosen.

Changing rows is simple, but not the only approach. It is also possible to first solve the problem of representing all boundary points in terms of inner points using the boundary conditions, and then subsequently solve the inner problem. This is often a waste of (coding) time in trade for a small computational gain. Nonetheless, many people implement code in this way and so we present the approach (which is a simple linear algebra exercise). We thus consider an already discretised equation

$$\mathbb{A}\boldsymbol{f} = \boldsymbol{b}, \tag{4.39}$$

where we have not replaced rows using our boundary conditions. $\mathbb{A}$ is an $N \times N$ matrix, where N is number of points in our grid.

Boundary conditions, be they Dirichlet or Neumann can be written as their own system of equations

$$\mathbb{B}\boldsymbol{f} = \boldsymbol{\beta}, \tag{4.40}$$

where $\mathbb{B}$ is an $M \times N$ matrix, M being the number of boundary conditions, the values of which are stored in $\boldsymbol{\beta}$. For each boundary condition, a corresponding point is chosen (the row we would have replaced). The vector $\boldsymbol{f}$ is then reordered such that these are the last M entries, and $\mathbb{A}$, $\mathbb{B}$ and $\boldsymbol{b}$ are permuted similarly. We now need to solve for the remaining $R = N - M$ points.

After the reordering, we can split Eq. (4.39) and write it as

$$\begin{pmatrix} \mathbb{A}_{RR} & \mathbb{A}_{RM} \\ \mathbb{A}_{MR} & \mathbb{A}_{MM} \end{pmatrix} \begin{pmatrix} \boldsymbol{f}_R \\ \boldsymbol{f}_M \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_R \\ \boldsymbol{b}_M \end{pmatrix}, \tag{4.41}$$

from which we need the first R equations:

$$\mathbb{A}_{RR}\boldsymbol{f}_R + \mathbb{A}_{RM}\boldsymbol{f}_M = \boldsymbol{b}_R \tag{4.42}$$

Likewise for Eq. (4.40),

$$\begin{pmatrix}\mathbb{B}_R & \mathbb{B}_M\end{pmatrix}\begin{pmatrix}\boldsymbol{f}_R\\ \boldsymbol{f}_M\end{pmatrix} = \boldsymbol{\beta}, \tag{4.43}$$

which can be rewritten as

$$\boldsymbol{f}_M = \mathbb{B}_M^{-1}(\boldsymbol{\beta} - \mathbb{B}_R\boldsymbol{f}_R). \tag{4.44}$$

Here we have used the boundary conditions to express the chosen $\boldsymbol{f}_M$ in terms of the remaining $\boldsymbol{f}_R$.

Using these together, we finally find

$$\left(\mathbb{A}_{RR} - \mathbb{A}_{RM}\mathbb{B}_M^{-1}\mathbb{B}_R\right)\boldsymbol{f}_R = \boldsymbol{b}_R - \mathbb{A}_{RM}\mathbb{B}_M^{-1}\boldsymbol{\beta} \tag{4.45}$$

which is precisely R equations for R unknowns. Written differently, we have derived

$$\mathbb{A}_{\mathrm{bc}} = \mathbb{A}_{RR} - \mathbb{A}_{RM}\mathbb{B}_M^{-1}\mathbb{B}_R, \tag{4.46}$$

$$\boldsymbol{b}_{\mathrm{bc}} = \boldsymbol{b}_R - \mathbb{A}_{RM}\mathbb{B}_M^{-1}\boldsymbol{\beta}. \tag{4.47}$$

Notice that $\mathbb{B}_R = 0$ if we only have Dirichlet boundary conditions in which case only $\boldsymbol{b}$ needs to be updated. In this case $\mathbb{B}_M^{-1}$ is also trivial to calculate as it will typically be equal to the identity matrix.

4.4 Spectral Method

Spectral methods are really powerful also for PDEs. They can be implemented for all types of boundary conditions and geometries for which suitable analytical basis functions exist.

We will only present an example for a square domain with periodic boundary conditions, as for these we can use Fourier series and the fast fourier transform. In these cases we can write our function to be solved for as

$$f(\boldsymbol{x}) \approx \sum_{\boldsymbol{k}} c_{\boldsymbol{k}}\, e^{i\,\boldsymbol{k}\cdot\boldsymbol{x}}. \tag{4.48}$$

For instance, in two dimensions

$$f(x,y) \approx \sum_{k_x=-N}^{N} \sum_{k_y=-N}^{N} c_{k_x,k_y}\, e^{i(k_x\,x + k_y\,y)}. \tag{4.49}$$

Let us consider a concrete PDE

$$\nabla^2 f(x,y) + \eta\,\partial_y f(x,y) = g(x,y) \tag{4.50}$$

with periodic boundary conditions. After a Fourier transform we have

$$-(k_x^2 + k_y^2)\tilde{f}(k_x,k_y) + i\,\eta\,k_y \tilde{f}(k_x,k_y) = \tilde{g}(k_x,k_y) \tag{4.51}$$

Thus we can solve the discretised version of the PDE using the fast fourier transform as

$$\boldsymbol{f} = \mathrm{ifft2}\left(\frac{\mathrm{fft2}(\boldsymbol{g})}{i\eta\boldsymbol{k}_y - (\boldsymbol{k}_x^2 + \boldsymbol{k}_y^2)} \right), \tag{4.52}$$

where fft2 and ifft2 are the two-dimensional fast fourier transforms.[10] Using the spectral method it is thus no harder to solve

[10] We again recommend to use the real version of these if you have access to these. They will typically be called rfft2 and irfft2.

a PDE than an ODE as presented in Sec. 3.2.3. We refer to that section for implementation details.

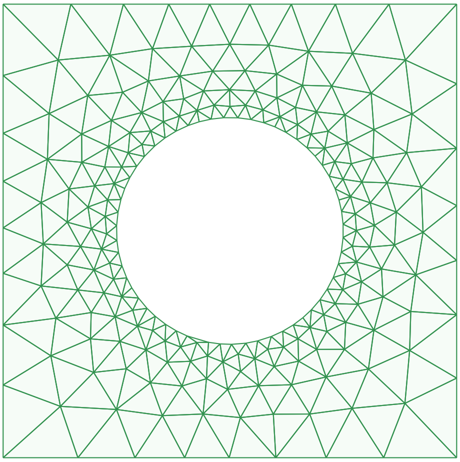
4.5 Finite Element Method

The finite difference method is in some sense the most "intuitive" method for discretising a differential equation. It is not the only method, however. If you are getting serious about numerical solutions to PDEs, you will very quickly stumble into the finite element method (FEM). FEM is slightly harder to introduce, but in the most basic version, it is far from as difficult as many resources make it seem. As it is very likely that you will stumble into this method at some point we include it here despite being an introductory text. The derivation, however, is somewhat longer than for the other methods presented in this text. We do not recommend to implement your own version of FEM as many great tools exist. But for using these tools, it is extremely useful to understand what is happening under the hood.

While FEM is only really useful for PDEs, it certainly can also be applied to ODEs. Our derivation for FEM will be done for a 2D PDE, however, it can instructive to compare this derivation with a 1D example. As such, we present FEM applied to an ODE on page 93. It can be useful to read this example before a second read-through of the derivation we present here for PDEs. The approach is very similar but the mathematical details slightly easier for ODEs.

We begin with some motivation. So far we have not considered

PDEs on unstructured grids. Unstructured grids such as



can be used to model special geometry, or enable higher accuracy to be obtained in certain regions. While finite difference schemes certainly can be derived on unstructured grids using multi-dimensional Taylor expansions, it is not straightforward. In particular it is very hard formulate high-order scheme of high accuracy using finite differences. In the finite element method, unstructured grids are as easy to handle as regular grids. And obtaining higher-order schemes is a lot more straightforward.

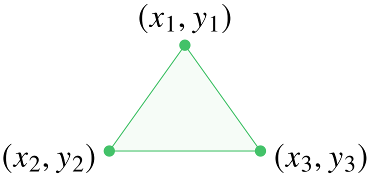
We are only going to present one of the simplest versions of FEM, and will only work with one example equation, namely Poisson's equation

$$\nabla^2 f(x, y) = g(x, y) \tag{4.53}$$

on a triangular mesh (i.e. on a grid made up of triangles). We use the notation in which the region on which we solve is denoted Ω and its boundary Γ. We will discuss how to implement both Dirichlet and Neumann type boundary conditions.

In some sense, the finite element method can be thought of as being a spectral method inside each triangle that is stitched together to find the total solution. In this sense, we can get some of the advantages of spectral methods, while having the possibility of local variations.

Consider a random triangle of our mesh:



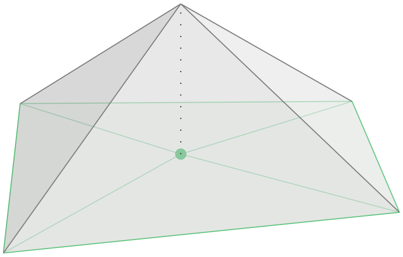
If we approximate, as we will, the function f to be linear inside each triangle, then the function is fully specified if we know the values of $f(x, y)$ at the triangle corners. For the above triangle, we thus simply need to solve for $f_1 = f(x_1, y_1)$, $f_2 = f(x_2, y_2)$, and $f_3 = f(x_3, y_3)$ to have its value everywhere inside the triangle.

We write the total function f as a sum of linear functions $\phi_i(x, y)$:

$$f(x, y) = \sum_i f_i\, \phi_i(x, y). \tag{4.54}$$

We have not yet specified what $\phi_i(x, y)$ are, other than to say that they are linear inside each triangle. Here comes the first of two crucial ideas that make up the finite element method: We choose the basis functions $\{\phi_i(x, y)\}$ such that they are equal to zero on all points of our mesh, except at a single point on which we require it to be equal to one (in the context of unstructured meshes, the points of the mesh are typically referred to as *nodes*). In this way we can associate each basis function to a specific point/node: The basis function ϕ_i belongs to node (x_i, y_i), and this is the only node on which it is not equal to zero. From this follows that Eq. (4.54) is the correct expansion if simply f_i is the value $f(x_i, y_i)$, i.e. the value of f at the node corresponding to ϕ_i.

How do we calculate these $\phi_i(x, y)$? We know they have to look like this:



Inside each triangle it is linear, and it goes from one to zero as we move away from the node to which it belongs. It is zero everywhere else. This is not too hard! Inside a triangle that has node i as a corner we must be able to write

$$\phi_i(x, y) = \alpha_1 + \alpha_2 x + \alpha_3 y. \tag{4.55}$$

To find the values of the α's we solve

$$\begin{pmatrix} 1 & x_1^i & y_1^i \\ 1 & x_2^i & y_2^i \\ 1 & x_3^i & y_3^i \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \tag{4.56}$$

where (x_1^i, y_1^i) is node that ϕ_i belongs to and (x_2^i, y_2^i) and (x_3^i, y_3^i) are the other corners of the triangle. This equation exactly enforces that $\phi_i(x_1^i, y_1^i) = 1$ and that it is equal to zero on the other corners.

Solving the equation we find

$$\begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \frac{1}{A} \begin{pmatrix} x_2^i y_3^i - x_3^i y_2^i \\ y_2^i - y_3^i \\ x_3^i - x_2^i \end{pmatrix} \tag{4.57}$$

where

$$A = \begin{vmatrix} 1 & x_1^i & y_1^i \\ 1 & x_2^i & y_2^i \\ 1 & x_3^i & y_3^i \end{vmatrix} = x_1^i y_2^i - x_1^i y_3^i - x_2^i y_1^i + x_2^i y_3^i + x_3^i y_1^i - x_3^i y_2^i$$

is determinant of the matrix (this equals twice the area of the triangle). Our basis function therefore evaluates to

$$\phi_i(x, y) = \frac{1}{A} \left[(x_2^i y_3^i - x_3^i y_2^i) + (y_2^i - y_3^i)\, x + (x_3^i - x_2^i)\, y \right] \tag{4.58}$$

inside this specific triangle. Note that ϕ_i will have different formulas inside different triangles and is non-zero only inside triangles that have node i as a corner.

You might be wondering how we are going to deal with the seconder-order derivative of Eq. (4.53) when we are representing f as a piecewise linear function. This is a good point! Indeed second derivatives of a linear function are equal to zero everywhere. This problem is solved with the second trick of the finite element method:

Multiply Eq. (4.53) by a function $h(x, y)$ and integrate over the entire domain to obtain

$$\int_\Omega \left(\nabla^2 f(x,y)\right) h(x,y)\,\mathrm{d}\Omega = \int_\Omega g(x,y)\,h(x,y)\,\mathrm{d}\Omega. \tag{4.59}$$

It should be pretty clear that the above equation holds for any choice of h if Eq. (4.53) holds, but why this is a useful thing to consider might be less obvious. In fact, if we *require* Eq. (4.59) to hold for *any* choice h, then Eq. (4.53) and Eq. (4.59) are exactly equivalent![11] This integral version [Eq. (4.59)] is called the *weak formulation* of the differential equation and is the one used in FEM. The function h is called a *test function*.

By integrating by parts, we obtain

$$-\int_\Omega \nabla f \cdot \nabla h\,\mathrm{d}\Omega + \int_\Gamma h\,\nabla f \cdot \hat{n}\,\mathrm{d}\Gamma = \int_\Omega g\,h\,\mathrm{d}\Omega, \tag{4.60}$$

where $\hat{n}$ is the unit normal vector to the boundary of our domain[12].

Neumann boundary conditions precisely specify the value of $\nabla f \cdot \hat{n}$ on the boundary. If we have any such boundary conditions

[11] We are not being very precise with defining which function spaces f and h belong to. But rest assured that the statement can be made precise.

[12] $\int_\Omega$ is an integral over our domain and $\int_\Gamma$ is an integral along the boundary of our domain.

we use them in the above expression. For boundaries where we have Dirichlet boundary conditions, the boundary integral terms will become irrelevant. Here we will assume that we either have only Dirichelet boundary conditions, or that our Neumann boundary conditions are of the form $\nabla f \cdot \hat{n} = 0$. This leaves us finally with the equation to be satisfied

$$-\int_\Omega \nabla f \cdot \nabla h \, \mathrm{d}\Omega = \int_\Omega g \, h \, \mathrm{d}\Omega \quad \text{for all } h. \tag{4.61}$$

It should now be clear how we can use linear functions to approximate f: we no longer have to deal with a second derivative. On our linearly approximated $f(x, y) = \sum_i f_i \phi_i(x, y)$, the equation becomes

$$-\sum_i f_i \int_\Omega \nabla \phi_i \cdot \nabla h \, \mathrm{d}\Omega = \int_\Omega g \, h \, \mathrm{d}\Omega \quad \text{for all } h. \tag{4.62}$$

To deal with the requirement ‘for all h‘, we will also expand h in our basis functions:[13]

$$h(x, y) = \sum_i h_i \, \phi_i(x, y). \tag{4.63}$$

With these linear approximations we can restate Eq. (4.62) as

$$-\sum_i f_i \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}\Omega = \int_\Omega g \, \phi_j \, \mathrm{d}\Omega \quad \text{for all } j, \tag{4.64}$$

[13] This is called the Galerkin method.

since if the equation holds for all basis functions, it will also hold for all combinations of them and therefore for any h.

We now simply need to perform the integrals, which only depend on quantities that are known, and then we have a linear equation for $\{f_i\}$ that can be solved by our usual approach.

You now know all of the main ingredients of the finite element method. What is left is the mathematics of carrying out the integrals. In principle, you could simply do numerical integration, but we can also do it analytically, which is naturally better. Let us explicitly evaluate the left-hand side of Eq. (4.64) for our linear basis function.

Clearly

$$\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}\Omega = 0 \tag{4.65}$$

if node i and node j do not share a triangle. If they do share a triangle, we have

$$\begin{aligned}
\phi_i(x,y) &= \frac{1}{A}\left[(x_2^i y_3^i - x_3^i y_2^i) + (y_2^i - y_3^i)\,x + (x_3^i - x_2^i)\,y\right], \\
\phi_j(x,y) &= \frac{1}{A}\left[(x_2^j y_3^j - x_3^j y_2^j) + (y_2^j - y_3^j)\,x + (x_3^j - x_2^j)\,y\right]
\end{aligned}$$

inside the shared triangle. Note that with our slightly cumbersome notation for the points, our formula is valid for both the case when $i = j$ and $i \neq j$.

Thus within the triangle

$$\nabla\phi_i \cdot \nabla\phi_j = \frac{1}{A^2}\left[(y_2^i - y_3^i)(y_2^j - y_3^j) + (x_3^i - x_2^i)(x_3^j - x_2^j)\right]$$

is constant and we find

$$\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}\Omega = \frac{1}{2A}\left[(y_2^i - y_3^i)(y_2^j - y_3^j) + (x_3^i - x_2^i)(x_3^j - x_2^j)\right] \tag{4.66}$$

since the integral of a constant over a triangle is simply equal to the constant multiplied by the triangle's area (= 1/2 A in our notation). More complicated terms can arise depending on the PDE under considering, but the integrals should nonetheless be fairly simple to carry out.

How to evaluate the right-hand side depends on $g(x, y)$. Numerical integration could be done directly on g, or one could expand g as well in the basis functions.

To summarise:

Finite Element Method

1. Rewrite the PDE in the weak formulation using a test function h
2. Choose basis functions $\{\phi_i\}$
3. Expand $f = \sum_i f_i \phi_i$ and find an equation for each j by taking $h = \phi_j$ in the weak formulation
4. Apply Neumann boundary conditions for the boundary integrals.
5. Evaluate all integrals to obtain a linear equation $\mathbb{A}\boldsymbol{f} = \boldsymbol{b}$

6. Apply Dirichlet boundary conditions changing the equation to $\mathbb{A}_{\text{bc}}\boldsymbol{f} = \boldsymbol{b}_{\text{bc}}$
7. Solve the linear equation to obtain $\boldsymbol{f} = (f_1, f_2, \cdots, f_N)$ which finally gives the solution $f(\boldsymbol{x}) = \sum_i f_i\, \phi_i(\boldsymbol{x})$

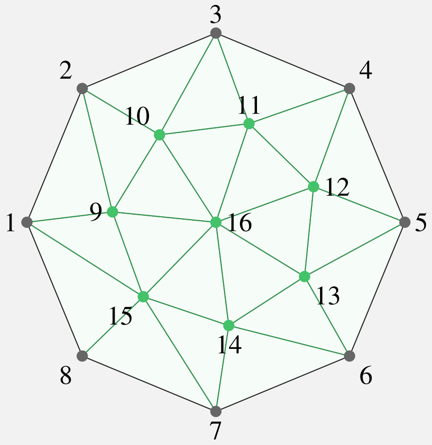
This was a lot of effort to describe the method. To really get a feeling for what we have developed, we give an example that illustrates all steps:

Example

Consider Laplace's equation

$$\nabla^2 f(x, y) = 0 \tag{4.67}$$

on the following 16-point unstructured mesh:



where grey colour indicates boundary nodes and edges.
We take the following boundary conditions:

$$\begin{cases} f(x,y) = \alpha & \text{on edge 1–2,} \\ f(x,y) = \beta & \text{on edge 6–7,} \\ f(x,y) \cdot \hat{n} = 0 & \text{on remaining boundary.} \end{cases} \tag{4.68}$$

Physically the solution of this boundary-value problem corresponds to the steady state temperature profile of the considered region, where the edge between node 1 and 2 is kept at temperature α, and the edge between node 6 and 7 is kept at temperature β, and no heat can escape through the rest of the boundary.
Taking $g = 0$ in Eq. (4.64), we write our equation as

$$\mathbb{A}\boldsymbol{f} = 0, \tag{4.69}$$

where for instance

$$\begin{aligned}
\mathbb{A}_{9,15} &= \int_{\Omega} \nabla\phi_{15} \cdot \nabla\phi_9 \, \mathrm{d}\Omega \\
&= \int_{T_{9,15,16}} \nabla\phi_{15} \cdot \nabla\phi_9 \, \mathrm{d}\Omega + \int_{T_{1,9,15}} \nabla\phi_{15} \cdot \nabla\phi_9 \, \mathrm{d}\Omega \\
&= \frac{1}{2A_{9,15,16}} \left[(y_9 - y_{16})(y_{15} - y_{16}) + (x_{16} - x_9)(y_{16} - x_{15}) \right] \\
&\quad + \frac{1}{2A_{1,9,15}} \left[(y_9 - y_1)(y_{15} - y_1) + (x_1 - x_9)(x_1 - x_{15}) \right]
\end{aligned}$$

using Eq. (4.66). Here $T_{i,j,k}$ refers to a specific triangle, and $A_{i,j,k}$ is the associated determinant. Note that there are two terms because node 9 and 15 share exactly two triangles. Likewise, for example, $\mathbb{A}_{16,16}$ will have seven terms, one for each triangle.

Our Neumann boundary condition $f(x, y) \cdot \hat{n} = 0$ is automatically applied at all boundaries due the partial integration [Eq. (4.60)]. Had we instead had $f(x, y) \cdot \hat{n} \neq 0$, we would have extra integrals to do. The Dirichlet boundary conditions are applied in the same manner as for the finite difference method.

Evaluating all integrals for our specific mesh we obtain

$$\mathbb{A}_{\text{bc}} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & -0.02 & 1.42 & 0.01 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & 0.01 & 1.39 & 0.01 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.01 & 1.41 & \cdots & -0.53 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
-0.12 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -1.43 & 0 \\
-1.33 & -0.29 & 0 & 0 & 0 & \cdots & 0 & 0 & -1.01 & -0.48 \\
0 & -1.13 & -0.49 & 0 & 0 & \cdots & 0 & 0 & 0 & -0.48 \\
0 & 0 & -0.93 & -0.68 & 0 & \cdots & 0 & 0 & 0 & -0.48 \\
0 & 0 & 0 & -0.73 & -0.88 & \cdots & -0.82 & 0 & 0 & -0.48 \\
0 & 0 & 0 & 0 & -0.53 & \cdots & 3.79 & -0.88 & 0 & -0.48 \\
0 & 0 & 0 & 0 & 0 & \cdots & -0.88 & 3.95 & -0.98 & -0.48 \\
-0.08 & 0 & 0 & 0 & 0 & \cdots & 0 & -0.98 & 4.10 & -0.48 \\
0 & 0 & 0 & 0 & 0 & \cdots & -0.48 & -0.48 & -0.48 & 3.37
\end{pmatrix},$$

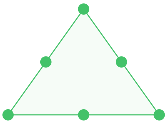
where on row $1, 2, 6, 7$ we have applied Dirichlet conditions. All rows sum to zero except those with Dirichlet conditions applied. Furthermore, we have $\boldsymbol{b}_{\text{bc}} = (\alpha, \alpha, 0, 0, 0, \beta, \beta, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T$.

Solving the linear problem

$$\mathbb{A}_{\text{bc}} \boldsymbol{f} = \boldsymbol{b}_{\text{bc}} \tag{4.70}$$

yields the solution at all nodes, which can then be interpolated using our linear basis functions.

We have only discussed the two-dimensional finite element method on triangles. The method can, naturally, be formulated in any dimension, and using more complicated meshes than triangular ones. The simplest extension to the method presented is to consider quadratic functions on triangular meshes. In this case, on each triangle six points are solved for in order to fix a quadratic polynomial[14] on the triangle. These are typically chosen as[15]



This is called the "P2-element" (or Lagrange Element of order 2), whereas we have presented FEM on the "P1-element" (Lagrange Element of order 1). In three dimensions, the simplest extension is linear functions on tetrahedral meshes. Modern FEM libraries will have all these versions, and many more, preimplemented.

As mentioned in the introduction, naturally, FEM can also be applied to ODEs, although their use case here is somewhat limited, as it is easy to obtain the same discrete formulations using the method of finite differences. Nonetheless we finish with an example illustrating FEM used to solve an ODE:

[14] $\phi_i(x, y) = \alpha_1 + \alpha_2 x + \alpha_3 y + \alpha_4 x^2 + \alpha_5 y^2 + \alpha_6 xy$

[15] We have to put three nodes on each edge in order to make the one-dimensional quadratic polynomials well-defined on the edge, which is shared between two triangles. Some element types will have nodes inside the triangles as well (e.g. P3-elements, which use cubic polynomials, will have a single node inside the triangles and four on each edge). Elements can also be defined that do not share any nodes between neighbouring triangles/cells. This can be useful if the solution is expected to have discontinuities.

Example

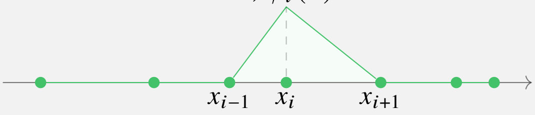
Consider

$$\frac{\partial^2 f(x)}{\partial x^2} = \gamma \tag{4.71}$$

for $x \in [a, b]$ with $f(a) = \alpha$ and $f'(b) = \beta$, γ is a constant. Multiplying by $h(x)$ and integrating by parts our weak formulation becomes

$$-\int_a^b f'(x)h'(x)\,\mathrm{d}x + [f'(x)h(x)]_a^b = \int_a^b \gamma h(x)\,\mathrm{d}x \quad \text{for all } h.$$

We will solve this on a grid $[x_1, x_2, \cdots, x_N]$ using piecewise linear basis function $\{\phi_i(x)\}$. We again choose these such that they are equal to 1 on their associated node and 0 elsewhere. Thus for instance, $\phi_i(x)$ looks like



which can be written as

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}} & x_{i-1} < x \le x_i \quad \& \quad i > 1 \\ \frac{x_{i+1} - x}{x_{i+1} - x_i} & x_i < x \le x_{i+1} \quad \& \quad i < N \\ 0 & \text{elsewhere.} \end{cases} \tag{4.72}$$

Writing $f(x) = \sum_i f_i \, \phi_i(x)$ and $h(x) = \phi_j(x)$, the equations to be satisfied for all j are

$$-f_i \int_a^b \phi_i'(x)\phi_j'(x)\,\mathrm{d}x = \begin{cases} ? & j=1 \\ \frac{1}{2}(x_{j+1}-x_{j-1})\,\gamma & 1<j<N \\ \beta & j=N. \end{cases}$$

where we used $f'(b)=\beta$, $\phi_N(b)=1$ and

$$\int_a^b \gamma\phi_j(x)\,\mathrm{d}x = \frac{1}{2}(x_{j+1}-x_{j-1})\,\gamma \tag{4.73}$$

for $1<j<N$. We have a question mark for $j=1$, because we do not have the value for $f'(a)$, but this does not matter, as the $j=1$ row will be replaced with the boundary condition $f(a)=\alpha$.

Evaluating the integral on the left-hand side we find for $i=j$:

$$\begin{aligned}\int_a^b \phi_i'(x)\phi_i'(x)\,\mathrm{d}x &= (1-\delta_{1i})\int_{x_{i-1}}^{x_i}\left(\frac{1}{x_i-x_{i-1}}\right)^2\mathrm{d}x \\ &\quad+(1-\delta_{Ni})\int_{x_i}^{x_{i+1}}\left(\frac{1}{x_{i+1}-x_i}\right)^2\mathrm{d}x \\ &= \frac{1-\delta_{1,i}}{x_i-x_{i-1}}+\frac{1-\delta_{N,i}}{x_{i+1}-x_i},\end{aligned} \tag{4.74}$$

where we used the Kronecker delta to also make our expressions valid at the boundaries. Likewise, for $j=i+1$ we find

$$\int_a^b \phi_i'(x)\phi_{i+1}'(x)\,\mathrm{d}x = \frac{\delta_{N,i} - 1}{x_{i+1} - x_i}. \tag{4.75}$$

All other integrals are equal to zero. We can now construct the matrix $\mathbb{A}$ which has entries $\mathbb{A}_{ij} = -\int_a^b \phi_i'(x)\phi_j'(x)\,\mathrm{d}x$, apply our boundary conditions and finally solve $\mathbb{A}_{\mathrm{bc}}\boldsymbol{f} = \boldsymbol{b}_{\mathrm{bc}}$ as usual.

On a regular grid with $x_{i+1} - x_i = \Delta x$, the matrix evaluates to

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{1}{\Delta x} & \frac{-2}{\Delta x} & \frac{1}{\Delta x} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\Delta x} & \frac{-2}{\Delta x} & \frac{1}{\Delta x} & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{\Delta x} & \frac{-2}{\Delta x} & \frac{1}{\Delta x} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{\Delta x} & \frac{-2}{\Delta x} & \frac{1}{\Delta x} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{-1}{\Delta x} & \frac{1}{\Delta x} \end{pmatrix} \boldsymbol{f} = \begin{pmatrix} \alpha \\ \gamma\Delta x \\ \gamma\Delta x \\ \vdots \\ \gamma\Delta x \\ \gamma\Delta x \\ \beta \end{pmatrix},$$

which is exactly what we found using finite difference if we divide the inner rows by Δx. We replaced the $j = 1$ row with the Dirichlet boundary conditions, which got rid of our question mark.

4.6 Non-linear Problems

The same exact ideas as presented for ODEs in Sec. 3.2.4 also apply for non-linear PDE problems. For instance, relaxation methods can readily be applied.

For some time-dependent problems, one can also apply a *semi-implicit* approach. Consider for instance the PDE

$$\frac{\partial f}{\partial t} = N(f)\, \nabla^2 f, \tag{4.76}$$

where $N(f)$ is some term that renders the equation non-linear, such as $N(f) = f$ or $N(f) = \sin(f)$. A fully implicit scheme for this equation is given by

$$f(t + \Delta t, \boldsymbol{x}) = f(t, \boldsymbol{x}) + N(f(t + \Delta t, \boldsymbol{x}))\, \nabla^2 f(t + \Delta t, \boldsymbol{x}). \tag{4.77}$$

This is a non-linear equation for $f(t + \Delta t, \boldsymbol{x})$, and requires e.g. relaxation methods to be solved.

A semi-implicit approach is given by

$$f(t + \Delta t, \boldsymbol{x}) = f(t, \boldsymbol{x}) + N(f(t, \boldsymbol{x}))\, \nabla^2 f(t + \Delta t, \boldsymbol{x}). \tag{4.78}$$

Now the non-linearity only depends on $f(t, \boldsymbol{x})$ and so the equation is linear in $f(t + \Delta t, \boldsymbol{x})$. The approach is called *semi*-implicit for the obvious reason that it is a mix of explicit and implicit terms, i.e. not fully implicit.

We end the next section with an example of a semi-implicit approach.

4.7 Operator Splitting

Consider a PDE

$$\frac{\partial f}{\partial t} = \mathcal{D}_1 f + \mathcal{D}_2 f, \tag{4.79}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are two spatial differential operators. An implicit scheme to solve this equation would be

$$f(t + \Delta t, \boldsymbol{x}) = f(t, \boldsymbol{x}) + D_1 f(t + \Delta t, \boldsymbol{x})\Delta t + D_2 f(t + \Delta t, \boldsymbol{x})\Delta t,$$

where D_1 and D_2 are the discretised versions of the differential operators. *Operator splitting* is an approach in which we solve the problem for each differential operator separately. In the present example, we could first solve

$$f^*(t + \Delta t, \boldsymbol{x}) = f(t, \boldsymbol{x}) + D_1 f^*(t + \Delta t, \boldsymbol{x})\Delta t, \tag{4.80}$$

for $f^*(t + \Delta t, \boldsymbol{x})$ and then subsequently solve

$$f(t + \Delta t, \boldsymbol{x}) = f^*(t + \Delta t, \boldsymbol{x}) + D_2 f(t + \Delta t, \boldsymbol{x})\Delta t \tag{4.81}$$

to complete the time step.

If we consider the right-hand side of the equation as physical forces, it is easy to get intuition for why this approach works: we simply first apply the effect of the first force and then apply the effect of the second force: the result will be approximately the same if Δt is small. When the forces applied are independent (in the sense that they commute) the method is exact. The order of the applying the operators can be further intermixed by taking two half-Δt time steps in turn for each operator. This is called Strang Splitting and will improve accuracy.

Operator splitting is particularly useful when solving systems of PDEs that depend on one another as it allows us to solve the time step of each equation independently of the others.

We finish this chapter with an example that showcases operator splitting as well as many of the approaches we have discussed in this chapter. In particular, we will demonstrate one approach to solving the Navier–Stokes equations. This text is introductory, and the Navier–Stokes equations are renowned for being difficult to solve numerically. The method we present will work in many cases, but obviously is not fit for all applications.

Example

Consider the incompressible Navier–Stokes equations

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} &= -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \nu \nabla^2 \boldsymbol{u} - \nabla p, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned} \tag{4.82}$$

Here, $\boldsymbol{u}$ is a velocity field and p a pressure field to be solved for. ν is the viscosity constant. In two spatial dimensions, where $\boldsymbol{u} = (u_x, u_y)$, this is a system of PDEs for three quantities: $u_x(t, x, y), u_y(t, x, y)$, and $p(t, x, y)$.
We could solve for all these quantities simultaneously, but this is quite tedious. Instead we will employ operator splitting

and semi-implicit time stepping which will allow us to solve for each quantity independently.

We employ operator splitting in which we start by ignoring the ∇p term. The tentative time step is then be performed by taking a semi-implicit approach, e.g.

$$\begin{aligned} u_x^*(t+\Delta t) &= u_x(t) \\ &\quad + \left[-(\boldsymbol{u}(t)\cdot\nabla)\, u_x^*(t+\Delta t) + \nu\nabla^2 u_x^*(t+\Delta t)\right]\Delta t \end{aligned} \tag{4.83}$$

and likewise for u_y^*. Note that $(\boldsymbol{u}(t)\cdot\nabla)$ is simply a linear differential operator as $\boldsymbol{u}(t)$ is known. It is thus no different than what we considered e.g. in Eq. (3.33).
We already know how to solve the PDE of Eq. (4.83), and we can use any method we wish: finite differences, spectral or finite element.

We now need to apply ∇p in the second part of our operator splitting step. Thus we need to evaluate e.g.

$$u_x(t+\Delta t) = u_x^*(t+\Delta t) - \partial_x p(t+\Delta t)\Delta t. \tag{4.84}$$

However, we do not yet know $p(t+\Delta t)$. This is set by the second part of Eq. (4.82).
If we take Eq. (4.84) and the corresponding equation for u_y we can inset $\boldsymbol{u}(t+\Delta t) = \big(u_x(t+\Delta t), u_y(t+\Delta t)\big)$ into $\nabla\cdot\boldsymbol{u}(t+\Delta t) = 0$ to obtain

$$\partial_x u_x^*(t+\Delta t) + \partial_y u_y^*(t+\Delta t) = \nabla^2 p(t+\Delta t)\Delta t, \tag{4.85}$$

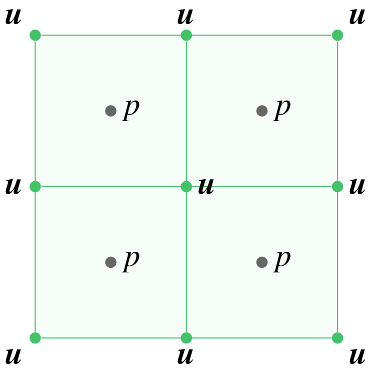
where we used $\nabla \cdot \nabla = \nabla^2$. But at this stage u_x^* and u_y^* are known and so this is simply a Poisson equation for $p(t + \Delta t)$ which we know well how to solve.

After solving for $p(t + \Delta t)$, we inset the solution into Eq. (4.84) and finally evaluate $\boldsymbol{u}(t + \Delta t)$. In this way each time step of the Navier–Stokes equation is taken by solving three equations: one for a tentative version of the velocity field $\boldsymbol{u}^*$, one for the pressure field p, and finally one for the real velocity field $\boldsymbol{u}$.

The method of the example is a version of *Chorin's projection method*. The advection term $(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$ moves the velocity field a distance $|\boldsymbol{u}|\Delta t$ each time step. We naturally run into problems if this is larger than our spatial discretisation (Δx). Thus we must choose Δt small enough to ensure this.[16] Other approaches to solving the Navier–Stokes equations avoid this issue.

Finally we note that the Navier–Stokes equations, and many other equations, are often solved on a staggered grid — i.e. grids where $\boldsymbol{u}$ and p do not live at the same points. For instance, we could discretise $\boldsymbol{u}(x, y)$ and $p(x, y)$ like this:

[16] This is called the Courant–Friedrichs–Lewy (CFL) condition.



Note that a central finite difference scheme of e.g. p in an equation for $\boldsymbol{u}$ is naturally formulated on such grids This is good for accuracy and helps avoid some numerical problems ("checkboard problems") when using non-projection methods to solve fluid problems. We will not discuss such approaches further here, but mention them only so you are aware of their existence.

Stochastic Systems

Not all physical laws are deterministic differential equations. Often we have to deal with stochastic systems. The source of randomness could be true random events such as nuclear decay or the measurement of a wave function, or it could be the seemingly random behaviour of stock markets or microorganism motility. Perhaps the most famous stochastic system is that of Brownian motion: the random movement of small particles due to thermal noise.

To able to simulate stochastic systems we need the computer to able to sample random numbers. For instance, the time t between events of radioactive decay is exponentially distributed

$$p(x) = \lambda e^{-\lambda t}. \tag{5.1}$$

So in order to simulate such a system, we need to be able to sample exponentially distributed numbers. Most programming languages provide random number generators for standard distributions, but custom methods are needed for special distributions. We begin this chapter with discussing this, and end with a few methods that are typically used for simulating specific random systems.

5.1 Random Numbers

To simulate random events on a computer, you need to be able to sample random numbers. However, a computer is a deterministic machine and cannot do anything truly random. All we can do is some mathematical operations that make it seem random enough. This is called pseudo-random number generation. To give a simple example, consider the sequence generated by[1]

$$x_{n+1} = 48271\, x_n \mod (2^{32} - 1). \tag{5.2}$$

We start with some *seed* x_0 and then keep applying the above formula. For instance, starting with $x_0 = 1656264184$ yields

$$x_1 = 3007196734, \quad x_2 = 3383877799, \quad x_3 = 1264039384 \cdots \tag{5.3}$$

In this way we can generate pseudo-random integers between 0 and $2^{32} - 2$. The initial seed x_0 could be taken from some source that constantly changes such as the time on the computer in microseconds, or similar. If we need random integers between 0 and some number N, we simply use $y_n = x_n \mod (N + 1)$.

In physics we are typically interested not in integers, but real numbers. If, for instance, we need random numbers sampled between 0 and 1 we could then simply take

$$r_n = \frac{x_n}{2^{32} - 2}, \tag{5.4}$$

which is a good approximation to a random uniform number.

The scheme we just presented is not great though. There are much better versions, and you will probably never need to implement your own.

[1] Recall that $a \mod b$ means the integer remainder of the division a/b.

It is good to understand, nonetheless, the principles behind such number generation. In this chapter we are going to assume that you have access to a library that reliably generates pseudo-random numbers. In particular, we will assume that you can generate integers, both uniform and Poisson distributed, and real numbers, both uniform and normal distributed.

5.1.1 Inverse Transform Sampling

Suppose you need to sample random numbers from a distribution $p(x)$, but this distribution is not implemented in your language of choice. This is in fact a very common situation. If the probability distribution is simple (and 1D), the best method to use, by far, is inverse transform sampling.

Inverse Transform Sampling

To sample from a probability distribution $p(x)$, solve for the inverse cumulative distribution $Q(u)$:

$$\int_0^x p(x')\,\mathrm{d}x' = u \Leftrightarrow x = Q(u). \tag{5.5}$$

Now sample a uniform random number U between 0 and 1. The number

$$X = Q(U) \tag{5.6}$$

will then be a random number sampled from $p(x)$.

Proving this method works is quite simple, although we will skip a formal derivation here. Intuitively, nonetheless, you can note that the cumulative distribution is always a function that maps an input x to a number between $[0, 1]$. We choose a random location on this y-axis and ask which x that corresponds to by using the inverse function.

Example

To sample a random number from the exponential distribution $p(x) = \lambda e^{-\lambda x}$ (defined on $[0, \infty]$) we first need solve

$$\int_0^x \lambda e^{-\lambda x} \, \mathrm{d}x' = 1 - e^{-\lambda x} = u \tag{5.7}$$

for x as a function of u. This one is easy and we find

$$Q(u) = -\frac{1}{\lambda} \log(1 - u). \tag{5.8}$$

Now we can use a standard sampler on a uniform interval to sample exponentially distributed numbers. Note that if U is uniform on $[0, 1]$ then so is $1 - U$, so we can also use

$$Q(u) = -\frac{1}{\lambda} \log(u). \tag{5.9}$$

Observe that indeed $Q(u)$ will map to $[0, \infty]$ for input in $[0, 1]$, as must be the case.

The formula can be used even if the equation cannot be solved analytically, as a numeric solution is adequate. In fact, not even the integral needs to be solved analytically, but helps in terms of speed of the algorithm.

5.1.2 Rejection Sampling

Inverse transform sampling works well for one-dimensional problems, but sometimes you need to sample for multi-dimensional distributions. For this rejection sampling can be used:

Rejection Sampling

To sample from a probability distribution $p(x)$, choose a proposal distribution $q(x)$ from which it is simpler to sample and which is non-zero for all x where $p(x)$ is non-zero.
Find an M (preferable as small as possible) such that

$$p(x) \leq Mq(x) \qquad \text{for all } x \tag{5.10}$$

Then

1. Sample an x from $q(x)$
2. Sample a uniform random number U on $[0, 1]$
3. - If $U \leq \frac{p(x)}{Mq(x)}$ keep the sample
   - Otherwise start over.

This method is also very simple to use, but how fast it is depends on the choice of $q(x)$. Preferably, q should be chosen to be as close to $p(x)$ as possible in order to avoid rejection by the last step. Intuitively, you expect to sample about M numbers before getting an acceptance. Therefore it is important to choose a q that minimizes M.

We will also skip a formal derivation of this method, but again it should be fairly intuitive: You sample from q and then adjust for the fact that this is the wrong distribution by making samples more unlikely in proportion to the distance $|p(x) - M\, q(x)|$.

Example

Consider sampling from the two-dimensional distribution

$$p(x, y) = \frac{1}{4\pi^2} \left(1 + \cos(x + y)\right) \tag{5.11}$$

for $x, y \in [0, 2\pi]$. As proposal distribution we simply choose the uniform distribution

$$q(x, y) = \frac{1}{4\pi^2}, \tag{5.12}$$

which is extremely easy to sample from, as we just sample two uniform random numbers on $[0, 2\pi]$. The maximal value of $p(x, y)$ is $\frac{1}{2\pi^2}$, and so the best M we can choose is $M = 2$.

5.1.3 Markov Chain Monte Carlo

Sometimes the probability distribution you are trying to sample from is too complicated for rejection sampling to work well. In this case you can turn to Markov Chain Monte Carlo (MCMC). The idea of MCMC is to throw away the requirement that we need independent samples. Instead we create a long sequence of samples where each sample is allowed to be correlated with the previous, but have the same statistics as independent samples if shuffled.

MCMC is a random walk in parameter space, biased in such a way that we spent more time in areas of high probability. We carefully choose the biasing such that after a long time, we have perfectly sampled the probability distribution.

We present only the simplest version of Markov Chain Monte Carlo, which depend on a choice of jump distribution $g(x\,|\,x')$ that is easy to sample from. For this one, if the parameters are continuous, we will often choose a Gaussian

$$g(x\,|\,x') = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-x')^2/2\sigma^2}. \tag{5.13}$$

MCMC: Metropolis–Hastings

To sample N points from $p(x)$ choose a jump distribution $g(x\,|\,x')$ from which it is easy to sample. Choose a starting point x_0. Then for $n \in [1, 2, \cdots N]$

1. Sample x from $g(x\,|\,x_{n-1})$.

2. Calculate $\alpha = \frac{p(x)}{p(x_{n-1})} \frac{g(x_{n-1} \mid x)}{g(x \mid x_{n-1})}$
3. - If $\alpha > 1$ set $x_n = x$
   - Otherwise sample a uniform random number U on $[0, 1]$.
     - If $U \leq \alpha$ set $x_n = x$
     - Otherwise set $x_n = x_{n-1}$.

For sufficiently large N, the sequence of samples can be shuffled to emulate independent samples of p.

Note that if g is symmetric, then $g(x_{n-1} \mid x)/g(x \mid x_{n-1}) = 1$, simplifying the method. This is for instance the case for Eq. (5.13).

The early samples of the sequence will depend on the choice of x_0. Therefore it is advisable to discard the first many samples (say the first 1,000, depending on the distribution being sampled). This is called the warmup or burn-in period.

The choice of g will massively affect the efficiency of the method. In the case that you use Eq. (5.13) for $g(x)$, how would you choose σ? For high-dimensional problems, the optimal choice is one that leads to acceptance probability of about 23 %. This can be tuned during warmup.

Note that Markov Chain Monte Carlo works even if you do not have access to a normalised distribution, as it only uses $p(x)/p(x')$. This is extremely useful both for physical simulations and for data modelling.

Physical simulation — As an example of using MCMC to do physical calculation we consider the Boltzmann distribution of statistical physics

$$p(\boldsymbol{x}) = \frac{e^{-E(\boldsymbol{x})/T}}{Z}. \tag{5.14}$$

Here, $\boldsymbol{x}$ is the micro-state of the system, E is the energy of the that state, T the temperature, and Z the partition function. The partition function is simply a normalisation, but is typically very hard to calculate. To calculate statistics of such a model, we can use MCMC to sample the distribution of microstates, without having to evaluate Z. In particular, if we use a symmetric jump distribution g, we simply have $\alpha = e^{-\Delta E/T}$. When used for physical simulations like this, the method is often referred to simply as *The Monte Carlo Method*, although this strictly speaking refers to the broad range of methods that use random number generation.

Data modelling — Consider the case, where you are trying to estimate some parameters x and y based on some data. From physical principles you derive[2] $p(\text{data} \,|\, x, y)$. This is typically what you can get from physics: if we already knew the values of x and y, we can calculate the probability of observing the data we did. From background information we also typically have a prior on x

[2] The notation $p(A|B)$ meaning the probability of A conditional on B having occurred.

and y: $p(x, y)$. Then from Bayes formula we have

$$p(x, y \,|\, \text{data}) = \frac{p(\text{data} \,|\, x, y)\, p(x, y)}{p(\text{data})}, \tag{5.15}$$

which is the function you want to sample from. The normalisation $p(\text{data})$ is hard to calculate, especially for high-dimensional problems, and so we typically only have access to

$$p(x, y \,|\, \text{data}) \propto p(\text{data} \,|\, x, y)\, p(x, y) = \mathcal{L}(x, y), \tag{5.16}$$

where $\mathcal{L}(x, y)$ is called the likelihood function. Fortunately, this is enough for MCMC to be able to sample x and y from $p(x, y \,|\, \text{data})$. It will furthermore typically be better to work in terms of log likelihoods to minimise floating points errors. In the case of symmetric g, the formula for α then becomes

$$\alpha = \exp(\log \mathcal{L}(x) - \log \mathcal{L}(x_{n-1})) \tag{5.17}$$

Often Markov Chain Monte Carlo is used to evaluate integrals of the form

$$I = \int_{-\infty}^{\infty} f(x, y)\, p(x, y)\, \mathrm{d}x\, \mathrm{d}y. \tag{5.18}$$

For instance, if $f(x, y) = x$, we calculate the mean value μ_x of x, and $f(x, y) = (x - \mu_x)^2$ will give the variance. To evaluate these integrals normally we would need a normalised probability distribution, but with Markov Chain Monte Carlo, we can estimate it as

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i, y_i), \tag{5.19}$$

where (x_i, y_i) are the sampled values of x and y. The error on the estimation of I will be of order $\mathcal{O}(N^{-1/2})$.

Finally, we note that Metropolis–Hastings is not the only MCMC method. More efficient methods (for continuous parameters) exploit knowledge of the gradient of $p(x)$. These are called Hamiltonian Monte Carlo methods and are beyond the scope of this text.

5.2 Event-based Simulations

The outcome of whether a coin falls heads or tails up is easy to simulate. We could simply sample one of the integers $\{0, 1\}$, and denote tails with zero and heads with one. This is perhaps the simplest form of an *event*-based simulation. If the coin is biased, e.g. tails happen with probability p, we could instead sample a uniform number U between [0, 1] and ask if $U < p$, in which case the toss is tails, and heads otherwise. This simple approach allows us to simulate event-driven systems. However, many physical systems are not formulated directly in terms of probabilities of events but instead in terms of *rates*. This section deals with how to simulate such stochastic systems.

An example of a stochastic systems specified in terms of rates is that of a chemical reaction such as

$$A + B \xrightarrow{k} C. \tag{5.20}$$

Here, A and B react together to produce C with rate k. If we have a large number of reactions, such system can readily be described by differential equations[3] as fluctuations will not be important. However, if we consider the reaction of a small number of reactants, fluctuations cannot be ignored.

For Eq. (5.20) let us denote the number of A reactants at time t by $N_A(t)$. Likewise, we define $N_B(t)$ and $N_C(t)$. The instantaneous total reaction rate will be equal[4] to $k\, N_A(t)\, N_B(t)$. At time t, how long until the next reaction occurs? By the very definition of rates, the chance that an event with rate R occurs in a short time interval Δt is $R\Delta t$. So the probability distribution of the time τ until next event is exponential:

$$p(\tau) = Re^{-R\tau}. \tag{5.21}$$

In the case of Eq. (5.20), $R(t) = k\, N_A(t)\, N_B(t)$. Finally, note that between events $R(t)$ is constant.

We now have all the ingredients to simulate an *exact* stochastic realization of Eq. (5.20). We simply sample a τ from Eq. (5.21), update time $t \leftarrow t + \tau$ as well as the molecular count $N_A \leftarrow N_A - 1$, $N_B \leftarrow N_B - 1$, and $N_C \leftarrow N_C + 1$, since after a reaction there will be one less A and B molecules, and one more C. This approach is called the Gillespie algorithm.

It is only slightly more complicated to simulate a system in which many type of events can occur. In the following we use

[3] In this case we could for instance have $c'(t) = k\, a(t)\, b(t)$.

[4] We are sloppy with the definition of k here, as it should be rescaled according to the volume of the system under consideration.

$\boldsymbol{x}(t)$ to denote the current state. For Eq. (5.20) this would be $\boldsymbol{x}(t) = (N_A(t), N_B(t), N_C(t))$.

The Gillespie Algorithm (Version 1)

Repeat until end of simulation:

1. Calculate current rates $\{r_i(t)\}$ using the current $\boldsymbol{x}(t)$.
2. For each event sample τ_i from $p(\tau_i) = r_i \exp(-r_i\,\tau_i)$.
3. Find the event corresponding to the minimum value sampled: $j = \arg\min_i\{\tau_i\}$.
4. Let $t \leftarrow t + \tau_j$ and update $\boldsymbol{x}(t)$ according to event j.

If the system being considered has a large number of events that can occur, it is slightly more efficient to use a different, but equivalent implementation:

The Gillespie Algorithm (Version 2)

Repeat until end of simulation:

1. Calculate current rates $\{r_i(t)\}$ using the current $\boldsymbol{x}(t)$.
2. Calculate the total rate $R = \sum_i r_i$.
3. Sample τ from $p(\tau) = R \exp(-R\,\tau)$.

4. Sample a uniform number U between 0 and R.

5. Find the first event j such that $\sum_{i=1}^{j} r_i \geq U$.

6. Let $t \leftarrow t + \tau$ and update $\boldsymbol{x}(t)$ according to event j.

This version only needs two random number samples, independent of the number of events. It is useful to know both of these methods as they are both often used. The two are equivalent because the distribution of the variable $X = \min(X_1, X_2, \cdots X_N)$ is exponential with parameter $R = r_1 + r_2 + \cdots r_N$ if X_i is exponentially distributed with parameter r_i.

The Gillespie algorithm simulates exact realizations of the stochastic rate equations. We recommend its use whenever it is possible. The only downside to the algorithm is that it becomes really slow if the rates are large, since the effective time step it takes will be of size $\Delta t \sim 1/R$.

In these cases we can turn to approximate methods, the simplest of which is called Tau-Leaping.

Tau-Leaping

Choose a time step Δt, and repeat until end of simulation:

1. Calculate current rates $\{r_i(t)\}$ using the current $\boldsymbol{x}(t)$.

2. For each event i, sample N_i from a Poisson distribution

with parameter $r_i \Delta t$:

$$p(N_i) = \frac{(r_i \Delta t)^{N_i} e^{-r_i \Delta t}}{N_i!}. \tag{5.22}$$

3. Let $t \leftarrow t + \Delta t$. For each event i, update $\boldsymbol{x}(t)$ by having the event occur N_i times.

We note that the method is called τ-leaping, because Δt is usually written using τ. We prefer Δt for consistency with the other methods, however. The above scheme is approximate, as we assume $\boldsymbol{x}(t)$ constant in the time between t and $t + \Delta t$, even though many events could occur in that time.

For differential equations we described the accuracy of a numerical scheme by how the error scaled with Δt. It is slightly harder to define the error for a stochastic simulation, since each time you run a simulation a different result will be found. This is the point of stochastic simulations after all. To define an error we ask what happens if we simulate many times and compare the average of such simulations to a true realization (such as one found by the Gillepsie algorithm). We have two choices for how to define this average error: we can take the error of the means or we can take the mean of the errors.

A method is described to have a *strong* order of convergence

$\mathcal{O}(\Delta t^n)$ if[5]

$$\max_t \mathbb{E}\,|X_{\text{sim}}(t) - X_{\text{true}}(t)| = \mathcal{O}(\Delta t^n), \tag{5.23}$$

where $\mathbb{E}$ denotes expectation (averaging over all simulations).

A method is described to have a *weak* order of convergence $\mathcal{O}(\Delta t^n)$ if

$$\max_t |\mathbb{E}[X_{\text{sim}}(t)^m] - \mathbb{E}[X_{\text{true}}(t)^m]| = \mathcal{O}(\Delta t^n) \tag{5.24}$$

for all integer values of m. Note that if Eq. (5.23) holds, then so does Eq. (5.24), but not the other way around.

Tau-leaping is order $\mathcal{O}(\Delta t)$ in the weak convergence, but only $\mathcal{O}(\sqrt{\Delta t})$ in strong convergence. You therefore need to use very small Δt when using this method. But very small can still be significantly larger than what is required by the Gillespie method for problems with large rates. Note that the definition of error is over the entire simulation, not per time step. Thus, the above should be compared e.g. to the total error of the Euler method of $\mathcal{O}(\Delta t)$ (since for ODEs the largest error will typically be found at the last time step $t = T$).

5.3 Stochastic Differential Equations

Event-based stochastic systems are discrete in time in the sense that there are finite periods of time over which nothing happens. In

[5] For simplicity we write maximum. To be mathematically precise this should a supremum.

contrast, Stochastic Differential Equations (SDEs) are the stochastic generalisation of Ordinary Differential Equations. Here we consider SDEs of the form

$$\mathrm{d}X = \mu(X,t)\,\mathrm{d}t + \sigma(X,t)\,\mathrm{d}W. \tag{5.25}$$

If you have never seen this notation before it can be a bit weird. Informally, you have think of $\mathrm{d}W$ as an infinitesimally small random number:

$$\mathrm{d}W = \lim_{\Delta t \to 0} \Delta W, \tag{5.26}$$

where ΔW is a normal distributed random number with mean zero and variance Δt. Note that this means that the standard deviation of ΔW is $\sqrt{\Delta t}$. Physicists often use the notation

$$\frac{\mathrm{d}X}{\mathrm{d}t} = \mu(X,t) + \sigma(X,t)\,\xi(t), \tag{5.27}$$

where $\xi(t)$ is a noise term. The former notation, however, is mathematically more well-defined and in fact also more natural for introducing numerical methods.

5.3.1 Initial-Value Problems

Just as for initial-value problems for ODEs, for SDEs we also choose a finite step size Δt. Almost as simple as the Euler Method, is the Euler–Maruyama method for SDEs:

Euler–Maruyama Method

Each time step is taken by updating

$$X(t+\Delta t) = X(t) + \mu(X(t),t)\Delta t + \sigma(X(t),t)\Delta W, \tag{5.28}$$

where ΔW is a normal distributed random number with mean zero and variance Δt:

$$p(\Delta W) = \frac{1}{\sqrt{2\pi\Delta t}} e^{-\Delta W^2/2\Delta t}. \tag{5.29}$$

The Euler–Maruyama method has weak order of error $\mathcal{O}(\Delta t)$, but only $\mathcal{O}(\sqrt{\Delta t})$ in strong order of convergence. When $\sigma(X,t)$ does not depend on X though, the strong order of convergence is $\mathcal{O}(\Delta t)$.

If σ does depend on X, a slightly better method which for all choices of σ has $\mathcal{O}(\Delta t)$ in strong order of convergence is:

Milstein method

Each time step is taken by updating

$$\begin{aligned} X(t+\Delta t) =& X(t) + \mu(X(t))\Delta t + \sigma(X(t))\Delta W \\ &+ \frac{1}{2}\sigma(X(t))\,\sigma'(X(t))\left(\Delta W^2 - \Delta t\right), \end{aligned} \tag{5.30}$$

where ΔW is a normal distributed random number with mean zero and variance Δt. Both occurrences of ΔW in Eq. (5.30) refer to the same random number.

Generalisations of Runge–Kutta methods to SDEs also exist (which do not use derivatives of σ), but these are beyond the scope of this text.

The SDE we have considered has been using the Itô integral. If you are aware of the distinction between Itô and Stratonovich integrals, you will know that Stratonovich SDEs are more common in physics than Itô. Conveniently, any Stratonovich SDE can be rewritten in the Itô interpretation by calculating the *noise-induced drift* term. After such a conversation the above methods can be applied. We note, nonetheless, that specific schemes designed for Stratonovich SDEs also exist.

5.3.2 Boundary-Value Problems

You are far more likely to run into stochastic initial-value problems than boundary-value problems. Nothing, however, prevents boundary-value problems from being well-defined for stochastic problems.

The classical example of this is the Brownian bridge: a random walk for which $X(0)$ and $X(T)$ are known. How would you in this case simulate $X(t)$? This specific case has a simple solution: simulate $\tilde{X}(t)$ considering only the initial-value boundary condition $X(0)$. Then

$$X(t) = \tilde{X}(t) + \frac{t}{T}\left(X(T) - \tilde{X}(T)\right) \tag{5.31}$$

is an exact realization of the equation, which satisfies both boundary conditions.

Most problems, however, will not have such an elegant solution. One approach to the general problem is to write down a likelihood function of the stochastic simulation. For the Brownian bridge example, we could for instance write

$$\mathcal{L} = \prod_{n=1}^{N} p(X(n\Delta t) - X((n-1)\Delta t)), \tag{5.32}$$

where $N\Delta t = T$. Markov Chain Monte Carlo methods, as presented in Sec. 5.1.3, can then be used to sample for $X(\Delta t)$, $X(2\Delta t) \cdots X((N-1)\Delta t)$. Note that if instead work in terms of log likelihood, as recommended in Sec. 5.1.3, the product in Eq. (5.32) becomes a sum.