Quest Journals
Journal of Research in Applied Mathematics
Volume 12 ~ Issue 11 (November 2026) pp: 34-46
ISSN (Online): 2394-0743 ISSN (Print): 2394-0735
www.questjournals.org

Research Paper

Numerical Methods in Physics A Foundational Guide to Differentiation and Differential Equations

Abdelaziz Mohamadain Elhaj Hussein [(1)] Badour Mohammed Abdleati [(2)]
(1) Secondary school teacher, Khartoum (2)Assikcte Professor, Nile Valley University, Faculty of Education

Abstract
This paper provides a comprehensive overview of essential numerical methods for solving physical problems, focusing on both foundational techniques and the rationale behind their usage. It begins with a detailed exploration of numerical differentiation, contrasting finite difference schemes (forward, backward, and central) and spectral methods using Fourier transforms, while critically analyzing their accuracy and the impact of machine precision on computational errors. The study then transitions to the numerical solution of ordinary differential equations (ODEs), systematically addressing both initial-value problems (via Euler, Verlet/Leapfrog, and Runge-Kutta methods) and boundary-value problems (using the shooting method and finite difference matrix formulations). The work emphasizes practical implementation, the importance of understanding underlying algorithms for troubleshooting and adaptation, and the utility of high-order schemes for efficiency and precision. The content is designed to equip readers with the necessary background to implement simple versions of these methods and to contextualize more advanced techniques encountered in academic literature and code.

Keywords: *Numerical Methods, Physics, Finite Difference, Spectral Methods, Ordinary Differential Equations (ODEs), Initial-Value Problems, Boundary-Value Problems, Runge-Kutta Methods, Shooting Method, Computational Physics, Machine Precision, Taylor Expansion.*

*Received 05 Nov., 2026; **R**evised 14 Nov., 2026; **A**ccepted 16 Nov., 2026 © The author(s) 2026.*
P**ublished with open access at **www.questjournals.org

I. Introduction:

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 finds 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 this work 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. 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.[2]

II. Numerical Differentiation and Its Role in Solving Physical Laws

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

First Order Derivatives (2.1) [4] 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.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 discretize the interval of interest by some small number Δx_1, and consider the function only at these points:

$$x \in \{a, a+\Delta x, a+2\Delta x, a+3\Delta x, \ldots, b-\Delta x, b\}. \quad (1)$$

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

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

This will only be truly equal in the limit of infinitesimally small Δx, but will be a good approximation also for reasonable small Δx. However, Eq.(2) is not the only possible choice we could take to approximate the derivative.

We could also do

$$f'(x) \approx \frac{f(x) - f(x-\Delta x)}{\Delta x}. \quad (3)$$

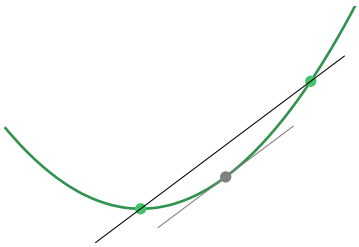
Which one of these two are the best? Intuitively, we expect both of these to give equally good approximations. In fact, there are many more choices. We could do something crazy like

$$f'(x) \approx \frac{f(x+2\Delta x) - f(x)}{2\Delta x}, \quad (4)$$

which indeed also tends to f ′(x) as Δx → 0. For finite Δx, however, this is a worse approximation than Eq. (2) and (3).

Eq. (2) and (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

$$f'(x) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}. \quad (5)$$



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

$$f(x-x_0) = \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(x_0)(x-x_0)^n = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2} f''(x_0)(x-x_0)^2 + \cdots \quad (6)$$

Let us try and use our finite difference formulas on these Taylor expansions. First, we use the forward scheme of Eq.(2). Without loss of generality, we evaluate the derivative for x = 0:

$$f'(0) \approx \frac{f(\Delta x) - f(0)}{\Delta x} = f'(0) + \frac{1}{2}f''(0)\Delta x + \cdot \tag{7}$$

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

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} + O(\Delta x). \tag{8}$$

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

Central Derivative

$$f'(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + O(\Delta x^2). \tag{9}$$

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.

Higher Order Derivatives (2.2) [4] 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. (9) on f′ we have

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

And then using the formula for x + Δx and x − Δx we have

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

Combining these we find

$$f''(x) = \frac{f(x + 2\Delta x) + f(x - 2\Delta x) - 2f(x)}{4\Delta x^2} + O(\Delta x), \tag{10}$$

which is a finite difference approximation for the second derivative.

Note that we used $O(\Delta x^2)/\Delta x = O(\Delta x)$, 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

$$f''(x) = \frac{f(x + \Delta x) + f(x - \Delta x) - 2f(x)}{\Delta x^2} + O(\Delta x^2). \tag{11}$$

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

Central Third Derivative

$$f'''(x) = \frac{1}{2\Delta x^3}\left[f(x + 2\Delta x) - 2f(x + \Delta x) + 2f(x - \Delta x) - f(x - 2\Delta x)\right] + O(\Delta x^2). \tag{12}$$

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 − 2Δx), f (x − Δx), f (x), f (x + Δx), and f (x + 2Δx), 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 $O(\Delta x^2)$:
Central Difference Coefficients with Accuracy $O(\Delta x^2)$:

	$-2\Delta x$	$-\Delta x$	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.

Deriving Schemes (2.3) [4] 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.
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 − 2Δx, x − Δx, x, x + Δx, x + 2Δx}.
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 as 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 General Formula for 5-Point Finite Difference Coefficients

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 \\ x_1^3 & x_2^3 & x_3^3 & x_4^3 & x_5^3 \\ x_1^4 & x_2^4 & x_3^4 & x_4^4 & x_5^4 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = d! \begin{pmatrix} \delta_{0,d} \\ \delta_{1,d} \\ \delta_{2,d} \\ \delta_{3,d} \\ \delta_{4,d} \end{pmatrix}. \tag{13}$$

Here δ is the Kronecker delta. The formula generalizes straightforwardly to smaller or larger stencils. The resulting scheme will have an accuracy of at least $O(\Delta x^{N-d})$, 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 (2.4) [4] 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 $\Delta x = 1$.

Machine Precision (4.5) [4] You might wonder why we need to develop all these different schemes. Even a method that has an error of the order $O(\Delta x)$ 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", the smallest representable number is about 10^{-18}. Using double-precision we can get down to around 10^{-321}. 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

$$1 + \epsilon_M = 1 + 2^{-52} \approx 1 + 2.2 \cdot 10^{-16}. \quad (14)$$

So what happens if you ask a computer to calculate $1+10^{-16}$? It will simply return 1. 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 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:

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

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)| ≈ 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$\sqrt{\delta}$, we only need to choose $\Delta x \sim \delta$. With this choice, the truncation error remains the same, but the rounding error is now only $\epsilon_M/2\sqrt{\delta}$. [11]

This is why higher-order schemes are useful: they allow us to use a larger Δx, which minimizes 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. 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

(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. 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.

Spectral Methods (2.6) [4] 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 $\{c_k\}$ and approximate

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

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

(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,

(17)

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

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

$$x \in \{0, \Delta x, 2\Delta x, 3\Delta x, \ldots, 2\pi - \Delta x\}, \quad (18)$$

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
Spectral Derivative using Fast Fourier Transform

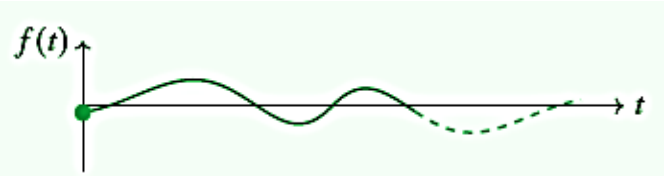
(19)

$$f^{(d)} \approx \text{ifft}\left((ik)^d\, \text{fft}(f)\right)$$

Here we assume that f (x) is stored as $f = [f_0, f_1, f_2, \ldots, f_n]$, 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. 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 is 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.
Spectral methods can becombined 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.

3. 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 $t = 0$.[6]



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

$$f'(t) = F(f(t), t) \quad (20)$$

and $f(0) = f_0$.
Here F is any (non-linear) function and f is the function to be found, both of which could be multi-dimensional. 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

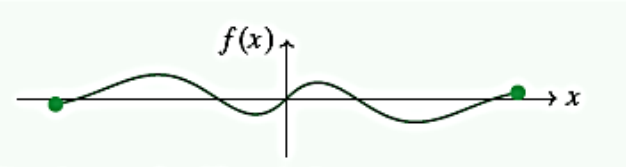
$$\begin{cases} f''(t) = F(f(t), t) \\ f(0) = \alpha \\ f'(0) = \beta \end{cases} \tag{21}$$

is equivalent to

$$\begin{cases} f_1'(t) = f_2(t) \\ f_2'(t) = F(f_1(t), t) \\ f_1(0) = \alpha \\ f_2(0) = \beta \end{cases} \tag{22}$$

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:[9]

Boundary-Value Problem



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

$$f'(x) = F(f(x)) \tag{23}$$

and n = dim(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

$$\begin{cases} f_1'(x) = f_2(x) \\ f_2'(x) = F(f_1(x), x) \\ f_1(a) = \alpha \\ f_1(b) = \beta \end{cases} \tag{24}$$

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 next.

Initial-Value Problems (3.1) [10] Choosing a step-size Δt to discretize time by, we can use the forward scheme of Eq. (8) to immediately give a method for solving initial-value problems:
The Euler Method

$$f(t + \Delta t) = f(t) + F(f(t), t)\,\Delta t \tag{27}$$

We know from Eq. (8) that the forward scheme has error of size $O(\Delta t)$, which means that each step in the Euler method has an error of size $O(\Delta t^2)$. 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 f (T) will be of order $T/\Delta t \times O(\Delta t^2) = O(\Delta t)$. Although the method is exceedingly simple to implement, this large error makes it unfit for many applications unless very small Δt are used.

Example (3.2) [10] Let us solve

$$f'(t) = \sqrt{f(t)}, \quad f(0) = 1 \tag{28}$$

using the Euler method with Δt = 0.1.

The first-time step gives us

$$\begin{aligned} f(\Delta t) = f(0.1) &\approx f(0) + \sqrt{f(0)}\,\Delta t \\ &= 1 + 0.1 = 1.1. \end{aligned} \tag{29}$$

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

$$x'(t) = v(t), \tag{29}$$

$$v'(t) = a(x(t)), \tag{30}$$

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

$$\begin{aligned} x_{\Delta t/2} &= x(t) + \frac{1}{2}v(t)\Delta t \\ v(t+\Delta t) &= v(t) + a(x_{\Delta t/2})\Delta t \\ x(t+\Delta t) &= x_{\Delta t/2} + \frac{1}{2}v(t+\Delta t)\Delta t \end{aligned} \tag{31}$$

The error of this method is only $O(\Delta t^3)$ 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.

Runge–Kutta Methods (3.3) [10] 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 $O(_{\Delta}t^2)$ per time step. A simple Runge–Kutta method to improve on this is Second Order Runge–Kutta (Midpoint Method)

For each time step update

$$\begin{aligned} k_1 &= F(f(t), t) \\ k_2 &= F\left(f(t) + \frac{1}{2}k_1\Delta t, t + \frac{1}{2}\Delta t\right) \\ f(t+\Delta t) &= f(t) + k_2\Delta t \end{aligned} \tag{32}$$

This method is strikingly similar to Verlet integration in that we do a half-Δt timestep. 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 (3.4) [10] Let us again solve

$$f'(t) = \sqrt{f(t)}, \quad f(0) = 1 \tag{33}$$

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

The first-time step is done by evaluating

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

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. (29)).
The most famous Runge–Kutta method is the simplest fourth order version:
Fourth Order Runge–Kutta (RK4), For each time step evaluate

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

The error of this method is $O(\Delta t^5)$ 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 +Δt) 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 $f(t+\Delta t)=f(t) + \Delta f_{RK}4 + O(\Delta t^5)$, whereas a fifth order method would have $f(t+\Delta t)= f(t) + \Delta f_{RK}5 + O(\Delta t^6)$. Thus, the difference
$E = |\Delta f_{RK}4 - \Delta f_{RK}5|$ will be of order $O(\Delta t^5)$. 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.
Boundary-Value Problems (3.5) [10] 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

$$f''(x) = g(x), \qquad x \in [a, b], \tag{34}$$

where g(x) is some specified function. The methods presented should generalize 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:

4. Dirichlet Boundary Condition

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

$$f(a) = \alpha \tag{35}$$

Neumann Boundary Condition

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

$$f'(a) = \alpha \tag{36}$$

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. (33) with the boundary conditions

$$f(a) = \alpha, \qquad f'(b) = \beta. \tag{36}$$

Shooting Method (4.1) [10] 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

$$f'(b) = \beta, \tag{37}$$

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

$$f(a) = \alpha, \qquad f'(b) = \gamma \tag{38}$$

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 fancier (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 generalize 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.

Finite Difference Method (4.2) [10] 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 discretize x, $f(x)$ and

$g(x)$ and store the value of these as vectors

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}, \qquad f = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}, \qquad g = \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_{N-1} \\ g_N \end{pmatrix}, \tag{39}$$

where $x_1 = a$ and $x_N = b$. Note that 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. (11), we have

(40)

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

$$f'' \sim f_{i+1} + f_{i-1} - 2f_i$$

$$f'' = \frac{1}{\Delta x^2} \begin{pmatrix} ? & ? & ? & ? & \cdots & ? & ? & ? & ? \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ ? & ? & ? & ? & \cdots & ? & ? & ? & ? \end{pmatrix} f \tag{41}$$

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 $O(\Delta x^2)$, we should do the same for the boundary condition. The backward scheme of second order is given by, $\frac{3}{2} f_N - 2f_{N-1} + \frac{1}{2}f_{N-2} = \beta\Delta x$.
All in all, the finite difference version of Eq. (33) with boundary conditions as given by Eq. (35) is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{2\Delta x} & \frac{-2}{\Delta x} & \frac{3}{2\Delta x} \end{pmatrix} \boldsymbol{f} = \begin{pmatrix} \alpha \\ g_2 \\ g_3 \\ \vdots \\ g_{N-2} \\ g_{N-1} \\ \beta \end{pmatrix}. \tag{42}$$

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 (4.3) [10] Use a finite difference scheme to rewrite the ODE as an approximate matrix equation,

$$A\, f = b. \tag{43}$$

For each boundary condition, choose a suitable row and change those rows in A and b with finite difference approximations to the boundary conditions. Finally solve

$$A_{bc}\, f = b_{bc} \tag{44}$$

to find f, where $_{bc}$ denotes the updated arrays.
Note that it is not necessary to invert A_{bc}, as faster methods exists that solve for $_f$ without doing explicit inversions. Also note that most entries of A_{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 (4.4) [10] Consider

$$f''(x) + h(x)\, f'(x) = 0,\ x \in [a, b] \tag{45}$$

with the same boundary conditions (35). We can again use the matrix of Eq.(41) for the second derivative. Let us call this A_2. In a similar fashion a first derivative matrix is

$$A_1 = \frac{1}{2\Delta x} \begin{pmatrix} -3 & 4 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -4 & 3 \end{pmatrix},$$

where we used Eq. (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. (45) is therefore,

$$A = A_2 + \mathrm{diag}(h)\, A_1, \tag{46}$$

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

Finally, note that if the equation had been

(47)

we would use

$$f''(x) + (h(x)f(x))' = 0, \quad x \in [a, b], \tag{48}$$

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 f. Our examples have used a regular grid-spacing of Δx. The method, nevertheless, works for irregular grid-spacing as well. When constructing A you can then use e.g. Eq. (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 function sexist 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 $A\,f = b$.

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

$$\lim_{f} |A\,f - b|^2, \tag{49}$$

i.e. find an f that minimizes the square distance between A_f and b. If the minimum is zero, we have found the solution. Writing $L = |A\,f - b|^2$, we have that the gradient of L with respect to f is

$$\nabla L = 2A^T (A\,f - b). \tag{50}$$

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

5. Conclusion

This work has presented a structured overview of essential numerical methods for solving physical problems computationally. Finite difference schemes offer simplicity, with central differences providing superior $O(\Delta x^2)$ accuracy, while spectral methods using Fourier transforms are critical for high-precision applications involving smooth, periodic functions. For initial-value problems, Runge-Kutta methods, particularly RK4, provide excellent accuracy, while Verlet integration offers symplectic energy conservation for conservative systems. Boundary-value problems are effectively addressed through the finite difference matrix formulation, which discretizes the entire domain and solves simultaneously, proving more robust and generalizable than the shooting method. Understanding truncation errors, stability, and machine precision limitations is essential for reliable method selection and interpretation of results.

References

[1] Press, W. H., Teukolsky, S. A., Vetterling, W. T.,&Flannery, B. P. (2007). Numerical Recipes: The Art of Scientific Computing (3rd ed.). Cambridge University Press. (Covers finite difference schemes, Runge-Kutta methods, and machine precision)
[2] Chapra, S. C.,&Canale, R. P. (2021). Numerical Methods for Engineers (8th ed.). McGraw-Hill Education. (Covers Taylor series, finite difference derivation, and ODE solvers)
[3] LeVeque, R. J. (2007). Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems. Society for Industrial and Applied Mathematics. (Covers finite difference methods for BVPs, matrix formulations, and boundary conditions)
[4] Butcher, J. C. (2016). Numerical Methods for Ordinary Differential Equations (3rd ed.). John Wiley&Sons. (Covers Runge-Kutta methods, error analysis, and adaptive step-size control)
[5] Trefethen, L. N. (2000). Spectral Methods in MATLAB. Society for Industrial and Applied Mathematics. (Covers spectral methods, Fourier transforms, and derivatives of periodic functions)
[6] Atkinson, K. E. (2008). An Introduction to Numerical Analysis (2nd ed.). John Wiley&Sons. (Covers numerical differentiation, truncation errors, and ODE boundary value problems)
[7] Iserles, A. (2009). A First Course in the Numerical Analysis of Differential Equations (2nd ed.). Cambridge University Press. (Covers Euler method, Runge-Kutta, and stability analysis)
[8] Hairer, E., Nørsett, S. P., & Wanner, G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems (2nd ed.). Springer. (Covers Runge-Kutta methods, Verlet integration, and symplectic methods)
[9] Quarteroni, A., Sacco, R., & Saleri, F. (2007). Numerical Mathematics (2nd ed.). Springer. (Covers finite difference coefficients, matrix construction for BVPs, and floating-point arithmetic)
[10] Mathews, J. H.,&Fink, K. D. (2004). Numerical Methods Using MATLAB (4th ed.). Pearson. (Covers Euler method, Runge-Kutta, and shooting method with practical examples)
[11] Kiusalaas, J. (2013). Numerical Methods in Engineering with Python 3. Cambridge University Press. (Covers finite difference schemes, boundary value problems, and matrix solvers)
[12] Stoer, J., & Bulirsch, R. (2002). Introduction to Numerical Analysis (3rd ed.). Springer. (Covers numerical differentiation error analysis, machine epsilon, and ODE methods)
[13] Higham, N. J. (2002). Accuracy and Stability of Numerical Algorithms (2nd ed.). Society for Industrial and Applied Mathematics. (Covers floating-point arithmetic, round-off errors, and machine precision)
[14] Boyd, J. P. (2001). Chebyshev and Fourier Spectral Methods (2nd ed.). Dover Publications. (Covers spectral methods for non-periodic domains and pseudo-spectral methods)
[15] Ascher, U. M., Mattheij, R. M. M.,&Russell, R. D. (1995). Numerical Solution of Boundary Value Problems for Ordinary Differential Equations. Society for Industrial and Applied Mathematics. (Covers shooting method and finite difference methods for BVPs in detail)
[16] Gautschi, W. (2012). Numerical Analysis (2nd ed.). Birkhäuser. (Covers finite difference derivation using Taylor expansions and interpolation)

[17] Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19(90), 297-301. (Original paper on the Fast Fourier Transform algorithm referenced in the spectral methods section)
[18] Verlet, L. (1967). Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. Physical Review, 159(1), 98-103. (Original paper on the Verlet integration method)
[19] Runge, C. (1895). Über die numerische Auflösung von Differentialgleichungen. Mathematische Annalen, 46(2), 167-178. (Original paper on Runge-Kutta methods)
[20] Kutta, W. (1901). Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. Zeitschrift für Mathematik und Physik, 46, 435-453. (Original paper extending Runge's work to higher-order methods)