SECTION 1
Course Description

Welcome to Computational Physics! This course is about the application of computational methods to solve mathematical problems in physics. In your core courses you have seen how physical systems can be described mathematically, often using differential equations, or statistically. Many of the examples you have encountered so far, in mechanics, electromagnetism and quantum physics, have simple analytic solutions. In real life the number of such problems is limited and **numerical methods** are used to solve most problems in mathematical physics, even for apparently simple systems.

In this course you will learn how to select and apply various techniques to solve mathematical physics problems, as well as how to test the suitability of the chosen numerical methods.

The skills acquired will be relevant for future work in both theoretical and experimental physics as well as mathematical modelling.

1.1. Overview

The course will cover the following topics.

- Solution of initial-value ordinary differential equations using finite difference methods.
- Analysis of the accuracy and stability of numerical methods for solving differential equations.
- Solution of initial-value parabolic and hyperbolic partial differential equations using finite difference methods.
- Linear matrix algebra. Iterative methods for solution of linear equations and eigenvalue problems.
- Use of matrix methods to solve elliptic (boundary-value) differential equations.
- Optimisation problems. Methods for finding the minimum of general multi-dimensional functions.
- How random number generators work and how to generate non-uniform random distributions.
- Monte Carlo methods for integration, minimisation and simulating equilibria of assemblies of particles.
- Fourier transform methods and their use in data processing and solving differential equations.
- An introduction to some library routines used to help solve numerical problems.

By the end of the course you should be able to:

- Select and implement finite difference methods to solve differential equations in physics.
- Evaluate the stability, accuracy and efficiency of a given finite difference method to solve a given differential equation.
- Formulate boundary value and eigenvalue problems as matrix equations.

- Select suitable random number generators and use them for simulations and integration.
- Design and implement Monte Carlo methods to simulate statistical physics problems.
- Show how Fourier transform methods can be used for data processing and solving differential equations.
- Design and write computer programs to solve physics problems using any of the above techniques.
- Use numerical library routines as *glass boxes* rather than *black boxes*.

Example problems will be drawn from a wide range of areas of physics such as mechanics, fluid dynamics, electromagnetism, quantum physics, statistical physics and high energy physics.

1.2. Course components

The course comprises the following components:

- A series of $15 + 1$ lectures, covering the theory of numerical methods and their applications in physics. (See table 1 for the lecture schedule.)
- Eight weekly practical sessions where you will apply the methods by undertaking ...
- ... two projects, where you will study two particular topics in depth.

Practical sessions will be held in the UG computing suite (level 3 Blackett) on **Wednesday mornings** from **9–12pm** during weeks 4–11 of term. We will use both the main and the teaching room. See table 2 for the schedule of practical sessions and project submission deadlines.

The course will be assessed in three units:

- **Project A**, to be submitted by **12 noon** on **Monday, 16th November 2015**. This will be worth **20%** of the marks for the course. There will be no choice for this project.
- **Project B**, to be submitted by the first Monday after the last day of term (**Monday, 21st December 2015** by **12 noon**). There will be a choice of four topics for this and the project will be worth **40%** of the marks.
- **Written Exam**, to be held in January 2016, worth **40%** of the marks. This exam will test your understanding of the numerical methods and algorithms, rather than involve writing code. The exam will take place on the first day of the 2nd term in the afternoon; **Monday, 11th January 2016, 2–4pm**. Make sure you are back at Imperial for the first day of the 2nd term and **allow for travel disruptions over the Christmas and New Year break as no exceptions will be allowed**.

You work individually on the coursework for projects A & B and submit a written report and your source code for each. The report covers the physics problem addressed, the numerical methods used to solve it, the results of your investigation, their analysis and interpretation and your conclusions. There is a 1,800 and 2,500 word limit for project A & B, respectively. All coursework will be assessed by a first & seconded marker.

Further details on coursework submission and assessment will be available in separate documents, later on in the course.

In addition to the assessed work, I will provide 4 problem sheets, which will not be assessed. They contain some theoretical and some programming problems to accompany and extend the lecture material, and examples of applications to physics. These problems will be relevant to the exam paper and to the projects. Exam past papers from the last few years are available on the course website on Blackboard Learn and the central Departmental UG Examinations web pages.

Table 1. Schedule of lectures (approx.) held in LT3

Topic	Lectures
Introduction to Numerical Calculations	2
Evaluating Finite Difference Methods	1.5
Methods for Integrating ODE Systems	1
Matrix algebra	1.5
Boundary Value Problems	1
Minimisation and Maximisation of Functions	1
Random Numbers & Monte Carlo Methods	1.5
Solving Partial Differential Equations - elliptical, hyperbolic, parabolic	3
Numerical Methods for Fourier Transforms	1.5
Other: course introduction & arrangements, case study, non-examinable examples.	*1*
Revision Lecture	*1*

Table 2. Practical Sessions and Project deadlines.

Lab	Date (in 2015)	Time	Topic
1	Wednesday, October 28th,	9am–12pm	Project A
2	Wednesday, November 4th,	9am–12pm	Project A
3	Wednesday, November 11th,	9am–12pm	Project A
*	**Monday, November 16th,**	**12 noon**	**Submission deadline project A**
4	Wednesday, November 18th,	9am–12pm	Project B
5	Wednesday, November 25th,	9am–12pm	Project B
6	Wednesday, December 2nd,	9am–12pm	Project B
7	Wednesday, December 9th,	9am–12pm	Project B
8	Wednesday, December 16th,	9am–12pm	Project B
*	**Monday, December 21st,**	**12 noon**	**Submission deadline project B**

1.3. Course website

The course website is **Computational Physics (2015-16)** on **Blackboard Learn** https://bb.imperial.ac.uk . Lecture notes, course handouts, problems sheets & solutions and reference material for coding will be made available there. The projects (consisting

of the report and the associated programs) should be submitted electronically though the course website on Blackboard Learn. Coursework will be checked by anti-plagiarism software; TurnItIn for the written report and a code similarity checker for the source code.

1.4. Project B

Four projects will be offered. The options will be reviewed briefly in lecture 12 and relatively detailed scripts will be provided for each. You may also propose your own project provided that it relates to the course material. If you have a clear idea of a project to study please discuss it with me, well in advance. The offered projects are:

1. Optimisation using log likelihood fit to find the lifetime of the D^0 meson.
2. Solution of Laplace's equation to study tolerances in misaligning capacitor plates for the electrostatic field.
3. Solution of the wave equation to study the dynamics of solitons.
4. Metropolis Monte Carlo simulation/integration for a statistical physics problem.

Further details on each project will be available in November.

1.5. Practical sessions

Unlike first and second year computing, you will be working much more on your own and in a less structured way. The practical sessions on Wednesday mornings are organised to give you access to help with programming problems and other questions. However, in practice you will need to be prepared to sort out most syntax errors on your own. You are not required to attend the practical sessions; we are there to help when you have problems, not to check your attendance. Remember that you will get most out of the help available at the practical sessions if you prepare your questions in advance. You are encouraged to help each other with practical aspects of programming, such as debugging code and so on, but the work you hand in for assessment must be your own.

The standard platform for practical work is Python on the PCs in the computing suite. Use of your own laptop and programming environment is allowed. However you do so at your own risk; there is no guarantee that any demonstrator will be familiar enough with your set up to help you out of technical problems. It is important that you (re)acquaint yourselves with the program development-environment (e.g. Python on PCs in the computing suite, or the IDE on your laptop) before the first session.

1.6. Programming language

The choice of programming language that you use for the projects is your own (subject to a few caveats). However please contact me for approval if you plan to use a language other than Python, C++ or plain C. I imagine that the Imperial MSci/BSc students on this course will prefer to use Python, as this is what you learnt in years 1 & 2. (However, Imperial physics students who started in October 2011 or before were taught C++.) If you wish to work in another language, such as FORTRAN or Matlab, this will normally be acceptable but you must check yourself that the software is available. One caveat is that, unless indicated otherwise in the project script, you must implement numerical

methods yourself rather than using a solver (e.g. DE solver or function minimiser) provided by a math library or built into the 'language'. The course is not a programming course as such but I will, in different places, provide small bits of advice to improve overall programming style. The quality of programming will influence the assessment of the projects at a minor level.

For graphs and tables you can use whatever tool you are familiar with to make effective plots for your report, e.g., matplotlib, Matlab, GnuPlot, Origin or Excel. (Of course, make sure axes are labelled and that multiple curves are distinguishable and labelled by a legend or in the figure caption.)

1.7. Plagiarism

All the work that you submit for assessment – the prose in the report, the code, the results and plots – must be your own. Any help from your colleagues or others must be clearly acknowledged in your submitted work. Occurrences of plagiarism are taken very seriously and can have dire consequences. See the Departmental Policy on Plagiarism. We are aware that reports and code have been posted online by graduated students, and are sometimes passed directly on by previous cohorts. These sources are in the plagiarism detection systems. Do not risk copying from them! Unfortunately, a small minority of students have been caught out in recent years. Hopefully this will not happen this year.

1.8. Contact

I will be available for discussions and questions during the lab hours in the Computing Suite. During term 1, I will also run **office hours** at the following times; **Tuesdays 1–2pm** and **Wednesdays 1–2pm**. I can be found in Room 724 Blackett, (ext. 47637). If you cannot make the regular office hour please contact me (rj.kingham@imperial.ac.uk)

Table 3. Computational Physics Demonstrators and Project Assessors.

Name	Position	Group	Room	email
Kingham, Robert	Staff	PLAS	Blackett 724	rj.kingham@imperial.ac.uk
Egede, Ulrik	Staff	HEP	Blackett 523	u.egede@imperial.ac.uk
Bruneau, Nicolas	RA	SPAT	Huxley 719	n.bruneau@imperial.ac.uk
Cameron, Richard	PG	PLAS	Blackett 735	richard.cameron08@imperial.ac.uk
Ducout, Anne	RA	ASTRO	Blackett 1003A	a.ducout@imperial.ac.uk
Gillies, Ewen	PG	HEP	Blackett 536	ewen.gillies12@imperial.ac.uk
Hamm, Joachim	RA	CMTH	Blackett 901C	j.hamm@imperial.ac.uk
McMahon, Andrew	PG	CMTH	Bessemer B321	andrew.mcmahon13@imperial.ac.uk
Mejnertsen, Lars	PG	SPAT	Huxley 6M67A	lars.mejnertsen10@imperial.ac.uk
Niasse, Nicholas	RA	PLAS	Blackett 739B	nniasse@imperial.ac.uk
Pusch, Andreas	RA	CMTH	Blackett 901	a.pusch11@imperial.ac.uk
Saba, Matthias	RA	CMTH	Blackett 816	m.saba@imperial.ac.uk
Sasihithulu, Karthik	RA	CMTH	Blackett 816	k.sasihithulu@imperial.ac.uk
Schmit, Claude	PG	ASTRO	Blackett 1014	claude.schmit13@imperial.ac.uk

to arrange an alternative. Seeing me in person is by far the most effective way of getting help on the topics covered by the course. Compared to an email exchange, it ensures that I properly understand the question, I can go through the necessary mathematics with you to answer it and I can illustrate my answer.

You can also contact me by email on rj.kingham@imperial.ac.uk. If you identify errors or other problems with the course material, or have general comments or suggestions, please let me know by email. The lab demonstrators are listed in table 3, please make use of them. If you need help outside the lab hours you can contact them as well but be prepared to be flexible about when to meet up to discuss your issue.

1.9. Literature

This course is bespoke and does not exactly follow any one textbook. However the following textbooks are strongly recommended:

- C. Gerald and P. Wheatley, *Applied Numerical Analysis*, International Edition, 7th edition, (Pearson, 2004), ISBN 0-321-19019-X.
- W. H. Press, B. P. Flannery, S. A. Teukolsky, and W.T Vetterling *Numerical Recipes in C++: The art of scientific computing* (Cambridge: CUP 2007, 3rd ed.). The full text of the second edition is available on the internet at http://www.nr.com/ but the access is rather convoluted.
- J. D. Hoffman, *Numerical Methods for Engineers and Scientists*, 2nd ed., (Marcel Dekker, Inc., 2001), ISBN 0-8247-0443-6.

The following resources are also useful:

- N. J. Giordano and H. Nakanishi. *Computational Physics*, Second Edition, (Pearson 2006), ISBN 0-13-146990-8. Good as background for some projects.
- E. Süli and D. Mayers. *An introduction to Numerical Analysis*, Cambridge University Press, ISBN-13 978-0-521-00794-8. A very formal & advanced book.
- E. W. Weisstein, *MathWorld–A Wolfram Web Resource.* http://mathworld.wolfram.com. Mathematics reference.
- G. B. Arfken and H-J. Weber, *Mathematical Methods for Physicists*, (Academic Press Inc 20051992), ISBN 0-12-088584-0. A reference book for most mathematics a physicists will ever need.
- M. Galassi et. al., *GSL - GNU Scientific Library.* Recommended numerical library for those programming in C++ or C. It is installed on the PCs in the UG Computer Suite. Source code and full documentation available at http://www.gnu.org/software/gsl/.

You will find many online sources related to computational methods and their applications to physics. These can be helpful for understanding, but please remember that web material is in general not completely reliable. Also remember to reference any sources (books, reports, websites, etc.) that you use in writing your project reports.

Acknowledgements

These lecture notes are adapted from previous course notes developed by C. Contaldi and U. Egede.

SECTION 2
Introduction to Numerical Calculations

Outline of Section

- Nature of errors
- Numerical differentiation
- Ordinary Differential Equations (ODEs)
- Natural units
- Solving non-linear algebraic equations
- Basic interpolation

2.1. Numerical Accuracy and Errors

Review of Taylor series

A computer approximates continuous functions as a truncated series expansion. This is because a computer can only add, subtract and multiply numbers. We will also be approximating derivatives as truncated series. To do this we review some basic concepts of Taylor series.

The function $f(x)$ can be expanded around the point a to n^{th} order as

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + ... + \frac{1}{n!}f^n(a)(x-a)^n\,,$$

where $f^n(a)$ is the n^{th} derivative of the function evaluated at the point $x = a$. In fact the function can be written as an n^{th} order expansion plus a remainder term which sums up all the remaining terms to infinite order

$$\boxed{f(x) = \sum_{i=0}^{i=n} \frac{1}{i!} f^i(a)(x-a)^i + R_n(x)} \quad \text{where} \quad R_n(x) \equiv \sum_{i=n+1}^{i=\infty} \frac{1}{i!} f^i(a)(x-a)^i\,. \tag{2.1}$$

Using the **Mean Value Theorem** it can be shown that there is a value ξ which lies somewhere in the interval between x and a for which

$$\boxed{R_n(x) = \frac{1}{(n+1)!} f^{n+1}(\xi)(x-a)^{n+1} \qquad \text{where} \qquad (a \le \xi \le x)} \tag{2.2}$$

that is that the remainder can be written as the $(n+1)^{\text{th}}$ term of the expansion evaluated at th point ξ. This is a useful way of expressing the error in the truncated series expansion for what follows. Remember that in order to possess an n^{th} order Taylor expansion, the function has to be n times differentiable (and $n+1$ times, if we want the remainder term).

For us it will be more useful to expand functions as $f(x+h)$ around the point x. This is because we will be interested in the value of the function at the point $x+h$ where h is a *small* step away from the point x where the value of the function $f(x)$ is already known. In this case, substituting in $x = a + h$, the Taylor series can be written as

$$f(x+h) \approx f(x) + f'(x)\,h + \frac{1}{2}f''(x)\,h^2 + ... + \frac{1}{n!}f^n(x)\,h^n\,. \tag{2.3}$$

For functions of two independent variables, say x and y, the Taylor series is

$$f(x+h, y+k) \approx \sum_{i=0}^{i=n} \frac{1}{i!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^i f(x,y) \tag{2.4}$$

where the differential operator $(\ldots)^i$ is expanded by the binomial expansion, e.g., $\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^2 = h^2\frac{\partial^2}{\partial x^2} + 2hk\frac{\partial^2}{\partial x \partial y} + k^2\frac{\partial^2}{\partial y^2}$, operates on the function and is then evaluated at (x,y). The remainder term is similar to (2.2), in that it involves $(n+1)^{\text{th}}$ order partial derivatives of f evaluated at the point (ξ,η), where $x \le \xi \le x+h$ and $y \le \eta \le y+k$. Equation (2.4) can be generalised to more independent variables by adding the relevant partial derivatives (e.g. $\partial/\partial z$ or $\partial/\partial t$) into the $(\ldots)^i$ term and using a multinomial expansion.

There are a number of errors that can affect the accuracy and stability of a numerical code.

Truncation errors

Even the most precise computer evaluates functions approximately. This approximation can be compared to the truncation of a Taylor series. Most programming languages use some form of power expansion to approximate standard mathematical functions. For example the Taylor expansion of $\sin(x)$ is

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ ... \ + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ ... \,.$$

If we truncate the series at 3rd order then the leading truncation error will be of 5th order

$$\sin(x) \approx p_3(x) \equiv x - \frac{x^3}{3!}\,,$$

then

$$\epsilon(x) \equiv \sin(x) - p_3(x) \sim \mathcal{O}(x^5)\,. \tag{2.5}$$

In general, if a Taylor series for a function $f(x)$ around $x = a$ is truncated at n^{th} order the error is

$$\epsilon = \frac{f^{n+1}(\xi)}{(n+1)!}(x-a)^{n+1}\,, \tag{2.6}$$

where ξ lies somewhere between x and a.

Round-off errors

Even in the absence of any truncation error a round-off error is inherent to all digital computers. This is due to the fact that the 'floating point' format used by computers stores any number with only a finite precision. The numbers are rounded off to the closest value at the computer's precision. With the `Python` programming language, the intrinsic (built in) floating point numerical type '**float**' is accurate to 15 significant decimal digits of precision. (With the 'right' number, it can be accurate to 17 significant figures.) In `C++` there is both 'single' precision (accurate to 6–8 significant figures) and 'double' precision (accurate to 15–17 significant figures; same as Python's 'float'). A perhaps surprising example of the effect of rounding error is subtracting 1/9 away from 1, nine times, e.g., in `Python`;

```
n=9; x=1.0; dx=x/n
for i in range(n):
       x=x-dx
print x
```

Rather than obtaining zero, the output (on Mac OSX) is **-1.665e-16** to 3 d.p. This round-off phenomenon occurs because the value $1/9 = 0.111\ldots$ is only stored to about 16 significant figures For this reason, it is advisable to use integer variables for loop counters rather than floating point variables.

A computer actually stores numbers in binary representation

binary representation		decimal representation
0	=	0
1	=	1
10	=	2
11	=	3
100	=	4
101	=	5
110	=	6
111	=	7
1000	=	8
etc...		

(2.7)

32 and 64 bits are normally used to store numbers in single and double precision respectively with the following scheme, e.g. for the number 1.2345×10^{13}

(2.8)

	sign	mantissa	exponent
	$\pm$	.12345	13
single	1-bit	23-bits	8-bits
double	1-bit	52-bits	11-bits

The exponent is stored as an 8-bit binary value ranging between -127 and +128 for single precision and an 11-bit binary value ranging from -1023 to +1024 for double precision. Thus the range of numbers allowed are roughly $10^{\pm 38}$ and $10^{\pm 308}$ for single and double precision respectively, as numbers are stored in base 2 (i.e. $2^{256/2} \sim 10^{38}$). Numbers smaller or greater than these will cause an **underflow** or **overflow** respectively. Overflows are replaced by the symbol `Inf` in some languages. The IEEE standard for handling, e.g., $0/0$, $\sqrt{-1}$, etc. is to replace with `NaN`, i.e., Not a Number.

Initial condition errors

Even in the absence of any truncation error or round-off error, an error in defining the starting point of the calculation can put the calculation onto a different solution of the equation being solved. This new solution (e.g. $x(t)$ curve) may diverge from the intended solution, perhaps more & more quickly with time (or whatever the independent variable is). Chaotic systems are particularly sensitive with respect to initial conditions, with the dynamics of Earth's atmosphere being a prime example. (Just think of the uncertainty of weather forecasting, particularly in the UK!)

Propagating errors

A propagation error is akin to an initial condition error. It is the error that would be seen in successive steps of a calculation given an error present at the current step, *if the rest of the calculation were to be done exactly (i.e. without truncation or round-off errors).* The **inherited error** at the current step is the accumulation of errors from all previous steps. Figure 2.1 illustrates propagating errors and truncation errors during each step of the numerical solution of an ODE $dy/dx = f(y, x)$. (Concrete examples of numerical schemes & ODEs will be given later.) The numerical scheme attempts to solve the ODE at discrete values of x; x_1, x_2, etc. You can see that the numerical solution (open circles) moves off the exact solution (solid line).

If the propagated error is increasing with each step then the calculation is **unstable**. If it remains constant or decreases then the calculation is **stable**.

The stability of a calculation can depend both on the type of equations involved and the method used to approximate the system numerically.

2.2. Numerical Differentiation

We are used to defining a derivative, e.g., dy/dx or $y'(x)$, as

$$\frac{dy(x)}{dx} \equiv \lim_{h\to 0} \frac{y(x+h) - y(x)}{h} \equiv \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}\,. \tag{2.9}$$

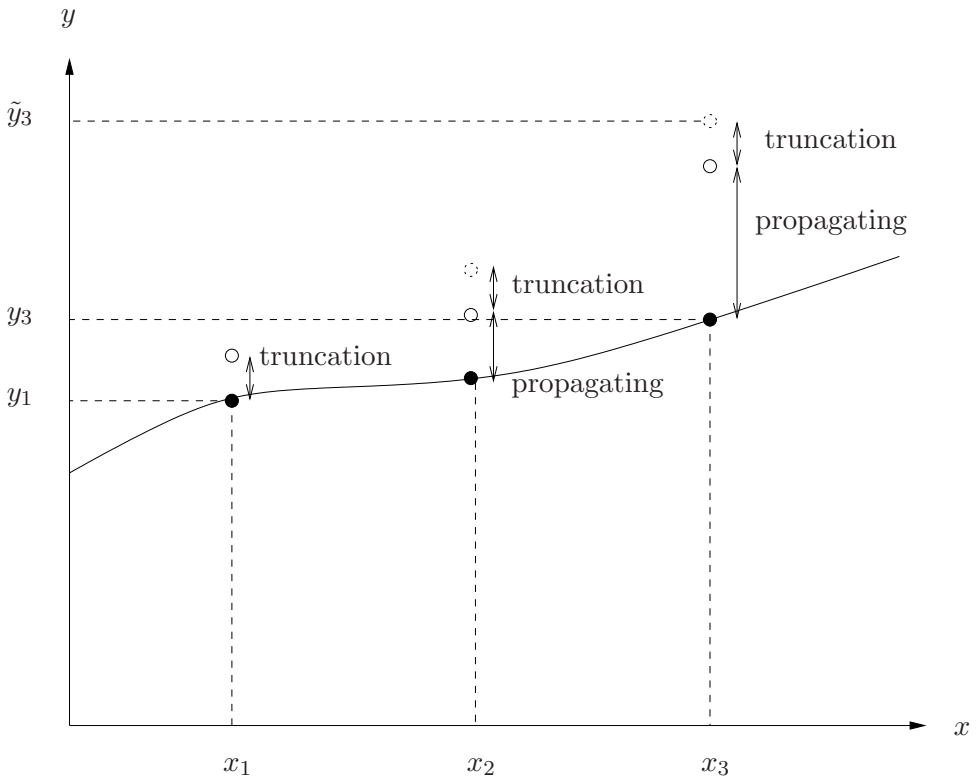

Figure 2.1. Illustration of truncation and propagating errors incurred in solving an ODE $dy/dx = f(y, x)$. The solid line depicts the exact solution. Dashed open circles depict the numerical solution. The calculation starts at (x_0, y_0) (off the plot) where there is no error. $\tilde{y}_3$ is the numerical result after 3 steps.

We approach this limit on the computer by defining a **forward difference scheme (FDS)**

$$\boxed{\tilde{y}'_f(x) \equiv \frac{y(x+h) - y(x)}{h}\,.} \tag{2.10}$$

The FDS is an example of a **finite difference approximation**.

Notation: in this course a tilde ($\sim$) over a quantity signifies that it is an approximated version, which is subject to truncation error. By considering the Taylor expansion of $y(x+h)$ around the point x we can understand how the error in the above scheme is first order;

$$y(x+h) = y(x) + y'(x)\,h + \frac{y''(\xi)}{2}\,h^2\,,$$

where we have defined the truncation error as

$$\epsilon = \frac{y''(\xi)}{2}\,h^2,$$

where ξ is an unknown value which lies in the interval $x < \xi < x+h$. Thus we can write the first derivative as

$$y'(x) = \frac{y(x+h) - y(x)}{h} - \frac{y''(\xi)}{2}\, h = \tilde{y}'_f(x) - \mathcal{O}(h)\,. \tag{2.11}$$

So the simple forward difference scheme has error $\mathcal{O}(h)$. Notice that even if we reduce this truncation error by making h really small, in practice, the accuracy of the derivative will have a limit imposed by the round-off error of the computer.

We can also use a **backward difference scheme (BDS)**

$$\boxed{\tilde{y}'_b(x) \equiv \frac{y(x) - y(x-h)}{h}\,,} \tag{2.12}$$

but as you can easily show (expand $y(x-h)$ around x) this will have a similar error to the forward difference estimate. However since they are estimates of the same quantity there must be a way to combine the two to get a more accurate estimate. This is achieved by the **central difference scheme (CDS)** which is the average of the two estimates and is 2nd order accurate,

$$\boxed{\tilde{y}'_c(x) \equiv \frac{\tilde{y}'_f(x) + \tilde{y}'_b(x)}{2} = \frac{y(x+h) - y(x-h)}{2h}\,.} \tag{2.13}$$

Consider the Taylor series (to 3rd order) for $y(x+h)$ and $y(x-h)$ giving

$$y(x+h) = y(x) + y'(x)\, h + \frac{1}{2} y''(x)\, h^2 + \frac{1}{3!} y'''(\xi)\, h^3 \tag{2.14}$$

and

$$y(x-h) = y(x) - y'(x)\, h + \frac{1}{2} y''(x)\, h^2 - \frac{1}{3!} y'''(\zeta)\, h^3 \tag{2.15}$$

then we can use this to define

$$y(x+h) - y(x-h) = 2\, y'(x)\, h + \mathcal{O}(h^3)\,,$$

which gives

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} - \mathcal{O}(h^2) \equiv \tilde{y}'_c(x) - \mathcal{O}(h^2)\,, \tag{2.16}$$

so the central difference scheme gives the derivative up to an $\mathcal{O}(h^2)$ error. Remember that h is a small step i.e. in suitable units (see later) it will satisfy the condition $h \ll 1$ so the error *decreases* with *increasing* order of magnitude in h.

You can understand why the CDS is more accurate than either FDS or BDS by looking at figure 2.2; the central difference gives a better estimate of the gradient (tangent line) at the point x than the forward or backward difference.

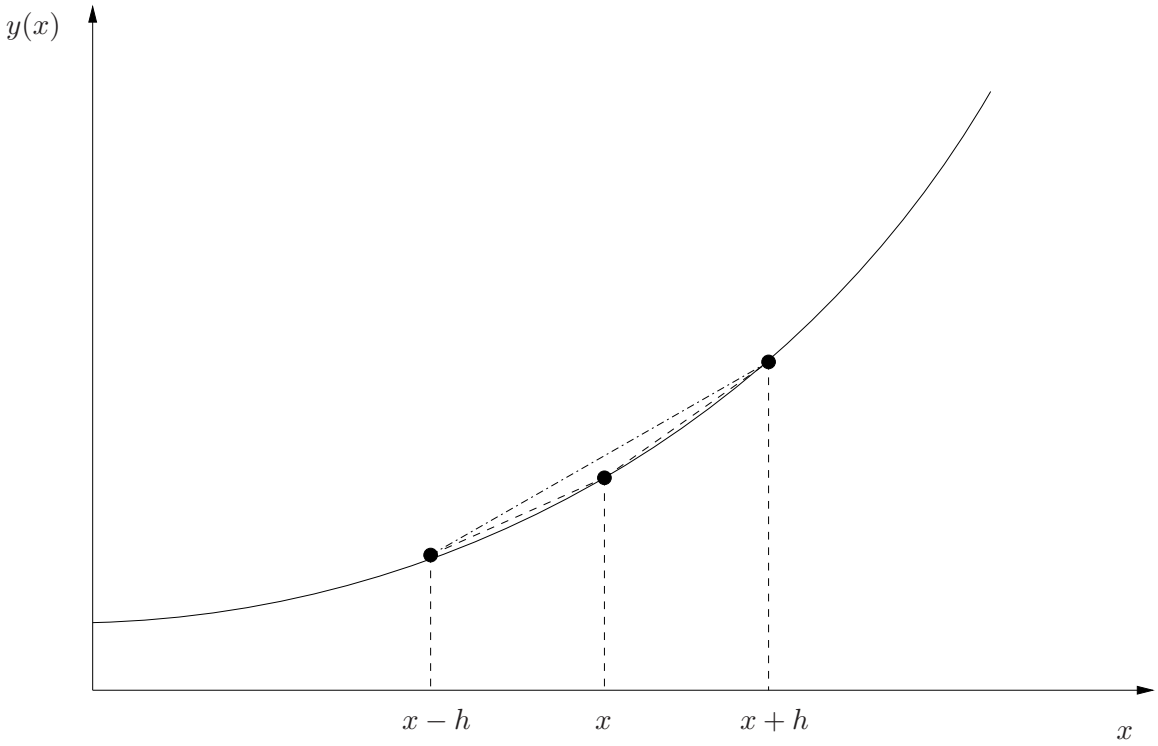

Figure 2.2. Forward, backward, and central difference schemes for approximating the derivative of the function $y(x)$ at the point x

Higher order derivatives

It is possible to find numerical approximations to 2nd and higher order derivatives too. A 2nd order accurate version of d^2y/dx^2 is

$$\tilde{y}''(x) \equiv \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = y''(x) + \mathcal{O}(h^2) \ . \tag{2.17}$$

This can be obtained by adding together the 4th order Taylor expansions for $y(x+h)$ and $y(x-h)$, i.e., equations (2.14) and (2.15) with one more term. Another way to get (2.17) is to take the central difference versions of 1st order derivative at $x+h/2$ and $x-h/2$, and then find the central difference of these;

$$\tilde{y}''(x) = \frac{\tilde{y}'_c(x+h/2) - \tilde{y}'_c(x-h/2)}{h} \ .$$

Three points, $y(x-h)$, $y(x)$ and $y(x+h)$, are needed to get $\tilde{y}''$, the finite difference approximation of y''. In general, to get the finite difference approximation $\tilde{y}^n$ requires at least $n+1$ points and going further away from x, e.g., $y(x+2h)$, $y(x-2h)$, $y(x+3h)$, $y(x-3h)$, etc.

2.3. Ordinary Differential Equations

In countless problems in physics we encounter the description of a physical system through a set of differential equations. This is because it is simpler to model the behaviour of a system (e.g. variables $\vec{y} \equiv \{u, v, w, \ldots, \text{etc.}\}$) in terms of infinitesimal responses to infinitesimal changes of independent variable(s) (e.g. η). The system will be of the form

$$\begin{aligned} \frac{du}{d\eta} &= f_u(\eta, \vec{y}) \\ \frac{dv}{d\eta} &= f_v(\eta, \vec{y}) \\ \frac{dw}{d\eta} &= f_w(\eta, \vec{y}) \\ &\vdots \end{aligned} \tag{2.18}$$

A compact notation for these coupled 1st order ODEs is

$$\frac{d\vec{y}}{d\eta} = \vec{f}(\eta, \vec{y}) \ , \tag{2.19}$$

where $\vec{f}(\eta, \vec{y}) = \{ f_u(\eta, \vec{y}), f_v(\eta, \vec{y}), f_w(\eta, \vec{y}), \ldots \}$, i.e., the 'components' of $\vec{f}$ are the functions in each ODE. Often the independent variable η is time t. Note that the functions defining the derivatives of the system variables are in general a function of the independent variable *and* all system variables, i.e., the system can be **coupled**. Only if the derivatives of each system variable are solely a function of that system variable is the system **uncoupled**.

The aim is to find a solution for the system variables at any value of the independent variable given a known initial condition for the system, e.g., solve for $u(\eta)$ given an initial condition $u(\eta_0)$ for all $\eta > \eta_0$.

To do this we need to integrate the differential equations. This can be done analytically for the simplest cases, e.g.,

$$\frac{dy}{dt} = \frac{t}{y} \qquad \text{with} \quad y(t=0) = 1 \, ,$$

then

$$\int_{y(0)}^{y(t_f)} y \, dy = \int_0^{t_f} t \, dt \, ,$$

giving

$$y(t_f) = \sqrt{t_f^2 + 1} \, .$$

Most often however, if the function describing the derivative is not separable or if the system of many variables is coupled, the system cannot be integrated analytically and we have to take a numerical approach.

We can solve systems numerically if the equations have a unique, continuous solution and we know the initial (or boundary value) conditions for the system variables.

Euler method

The simplest approach to numerical integration of an ODE is obtained by looking again at the Taylor expansion for a forward difference. Using time t as the independent variable,

$$\begin{aligned} y(t+h) &= y(t) + y'(t,y)\,h + \mathcal{O}(h^2) \\ &\approx y(t) + f(t,y)\,h\,. \end{aligned} \tag{2.20}$$

So if we know the value of the variable y at t we can use a finite step h to calculate the next value of y at $t+h$ up to $\mathcal{O}(h^2)$ accuracy. This Euler step can be iterated, stepping the solution forward h each time. In terms of the discretised limit of the function, and remembering that y is now an approximate solution (due to the truncation error incurred in dropping the $\mathcal{O}(h^2)$ terms), we can write this as

$$\tilde{y}_{n+1} = \tilde{y}_n + f(t_n, \tilde{y}_n)\,h \ , \tag{2.21}$$

where we use the subscript n to denote values of the function y at **time step**

$$t_n = t_0 + (n-1)\,h \ . \tag{2.22}$$

Notation: For brevity and convenience, the tilde ($\sim$) will normally be dropped when writing out such finite difference methods. (On occasion, when it is necessary to distinguish between actual/exact solutions and approximate solutions, $\sim$ symbols will be reinserted as appropriate.) Hence we chose to write the Euler method as

$$\boxed{y_{n+1} = y_n + f(t_n, y_n)\,h \ .} \tag{2.23}$$

Higher order systems

A system described by an m^{th}-order ordinary differential equation can always be reduced to a set of m 1st-order ODEs. For example, consider the 3rd-order, linear system

$$\frac{d^3y}{dt^3} + \alpha\frac{d^2y}{dt^2} + \beta\frac{dy}{dt} + \gamma\,y = 0 \ .$$

We can introduce three new variables

$$u \equiv y, \quad v \equiv \frac{dy}{dt}, \quad \text{and} \quad w \equiv \frac{d^2y}{dt^2}\,,$$

such that the system can be written as three 1st-order equations

$$\begin{aligned} \frac{du}{dt} &= v \\ \frac{dv}{dt} &= w \\ \frac{dw}{dt} &= -\left(\gamma\,u + \beta\,v + \alpha\,w\right)\,, \end{aligned}$$

or in matrix notation

$$\frac{d}{dt}\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -\gamma & -\beta & -\alpha \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix} .$$

Generalising further, a pair of coupled m^{th}-order and n^{th}-order ODEs can be reduced to a set of $m+n$ 1st-order coupled ODEs. (This can be extended to sets of coupled arbitrary order ODEs.) In general a **linear** ODE system can be written in terms of a matrix operator $\mathbf{L}$ as

$$\frac{d\vec{y}}{d\eta} = \mathbf{L}\,\vec{y} \ , \tag{2.24}$$

with initial conditions $\vec{y}(\eta_0) = \vec{y}_0$. The Euler method is then $\vec{y}_{n+1} = \vec{y}_n + h\,\mathbf{L}\,\vec{y}_n$ so that

$$\boxed{\vec{y}_{n+1} = (\mathbf{I} + h\mathbf{L})\,\vec{y}_n \equiv \mathbf{T}\,\vec{y}_n \ ,} \tag{2.25}$$

where $\mathbf{T}$ is the update matrix.

A general, non-linear system is written as

$$\frac{d\vec{y}}{d\eta} = \vec{f}(\eta, \vec{y}) \qquad \text{or alternatively} \qquad \frac{d\vec{y}}{d\eta} = \mathcal{L}(\vec{y}) \ , \tag{2.26}$$

where $\mathcal{L}$ is a **non-linear operator**. Non-linear systems cannot be expressed as linear matrix equations such as equation (2.24).

2.4. Natural units

It is very important to use the correct units when integrating systems numerically due to the limited range of numbers a computer can deal with. It also helps to understand the dynamics if we can scale the variables relative to appropriate characteristic units of measure, that encapsulate important physical properties of the system. This in turn greatly aids debugging.

The scaling process removes SI units from the variables (leaving them in 'natural' or 'dimensionless' units) and for this reason is also known as equation nondimensionalisation. Thus **natural units** , **scaling** and **nondimensionalisation** are all interchangeable terms. All the independent variables should be scaled. Scaling of the dependent variable is optional, but often insightful. Initial values or asymptotic values can be good choices. For example, consider the dimensionfull system for exponential decay with a source

$$\frac{dy}{dt} + \alpha\, y \ = \ g \tag{2.27}$$

where α is a decay rate and g a constant growth/source term. Assuming the dependent variable has SI dimensions $[y] = \text{m}$, then $[\alpha] = \text{s}^{-1}$ and $[g] = \text{m/s}$. We can then define a scaled time variable $\hat{t} = \alpha\, t$ such that [1]

$$\frac{d}{dt} = \frac{d\hat{t}}{dt}\frac{d}{d\hat{t}} = \alpha\frac{d}{d\hat{t}} \tag{2.28}$$

[1] There are many notations for scaled variables. Common alternatives include using a tilde, i.e., $\tilde{t}$ (which unfortunately clashes with our use of '$\sim$' to distinguish numerical approximations of things (derivatives, solutions of DE) from exact versions), using Greek symbols (i.e. τ) and using capitalised symbols (i.e. T).

and the equation becomes

$$\frac{dy}{d\hat{t}} + y \;=\; \frac{g}{\alpha} \equiv G \;\;, \tag{2.29}$$

where $[G] = \mathrm{m}$. We could chose to go further and scale y too. Given that the solution settles down to $y = g/\alpha = G$, a good choice is $\hat{y} = y/G$ so that

$$\frac{d\hat{y}}{d\hat{t}} + \hat{y} \;=\; 1 \;\;. \tag{2.30}$$

This is the original equation but with time scaled to the exponential decay time and the amplitude scaled to the final, steady-state value.

Changing to units where the independent variable is dimensionless means we don't have to worry about units in our integration;

- Step-size h; in these natural units a small step is anything with $h < 1$. If we still had dimensions in the problem this would be a lot harder to define.
- Range of integration; the characteristic timescale in the dimensionless units is $\hat{t} \sim 1$ so we know that integrating from $\hat{t} = 0$ to $\hat{t}_f$ (where $\hat{t}_f \sim$ a few) will cover the entire dynamic range of interest.

One thing we *must* remember is to put the SI units back in when we present our results, e.g., in plots of the integrated solution or explicitly state what natural units are (to give meaning to the numbers). For example, in a plot of $\hat{y}(\,\hat{t}\,)$ would label the horizontal axis in units of $[1/\alpha]$ and the vertical in units of $[g/\alpha]$.

2.5. Solving Non-Linear Algebraic Equations

An important part of the computational physics 'toolkit' is to find roots of non-linear algebraic equations, i.e., the solution of

$$f(x) = 0 \;\;,$$

where f involves transcendental functions or is even simply a polynomial of high order (i.e. $n > 4$), and therefore a solution in closed form is not possible. An example of a non-linear equation is $\sin(x^2) - 1/(x + a) = 0$. This is just the sort of situation where numerical methods are essential. Non-linear root finders are typically **iterative methods**, where an initial guess is refined iteratively. Functions with discontinuities and/or infinities pose problems unless the initial guess is carefully made. Some key methods are given below.

Treatment of many variables, e.g., $f(x, y, z) = 0$ will be covered in Section 7.

Bisection method

This is the simplest method and is very robust, but slow. One starts with two points x_l (the left point) and x_r (right point) which bracket the root. To bracket the root, $f(x_l)$ and $f(x_r)$ must have opposite sign. Then $f(x)$ must pass through zero (perhaps several times) between these points. Now get the mid point

$$x_m = \frac{x_l + x_r}{2}$$

and determine whether the pair x_l and x_m or x_m and x_r now bracket the root. Update x_l or x_r to x_m accordingly and repeat until $\epsilon_i = x_r - x_l$ (where subscript i denotes the iteration number) is sufficiently small or $\max[|f(x_l)|, |f(x_r)|]$ is sufficiently close to zero. (These are convergence criteria.) This method converges linearly, with the error ϵ_i (the uncertainty in where the root is) halving with each iteration;

$$\epsilon_{i+1} = \epsilon_i/2 \ .$$

Bisection will also find where discontinuous functions jump through zero.

Newton's method

Also known as the Newton-Raphson method. This iterative scheme uses the 1st derivative of the function

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \ .} \tag{2.31}$$

The Newton method can easily fail if it gets near maxima or minima, in which case x_{i+1} can shoot off 'to infinity'. It is also possible to be locked in a cycle where the method ping-pongs between the same two x points. The advantage of Newton's method is its speed of convergence; it converges quadratically,

$$\epsilon_{i+1} = \text{const} \times (\epsilon_i)^2 \ . \tag{2.32}$$

Secant method

This iterative methods is related to Newton's method, but works when an analytical expression for the derivative function $f'(x)$ is unknown. The tangent to the function $f'(x_i)$ is approximated using two points on the curve (which define the secant line)

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \ . \tag{2.33}$$

Inserting into Newton's method gives

$$\boxed{x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \ .} \tag{2.34}$$

The speed of convergence of the secant method is somewhere between linear and quadratic.

2.6. Basic interpolation

Finding a value for a function between points where it is known, is another essential piece of a computational physicist's toolkit. One approach is to fit a function in the least-squares sense to the data points, as is typically done with experimental data. This approximate fit will probably not go through any of the data points. Another approach, which is considered here, is to fit a polynomial that goes exactly through the points. This is interpolation.

Interpolation is an extensive subject. Here we briefly look at some very basic methods with which you can make do. (More advanced and powerful methods exist; see any text book!). Here is an example of when interpolation might be needed. In solving ODEs numerically, time is typically discretised (t_n). An idea of the solution at a time between these t_n might be required. As we will see later, when solving PDEs space is discretised into a grid and the numerical method provides the (approximate) solution at these points. Again, the solution at other points in space is often needed.

Linear interpolation

This is simply fitting a straight line through adjacent points (x_i, f_i) and (x_{i+1}, f_{i+1}) to find f at a point x in between;

$$\boxed{f(x) = \frac{(x_{i+1} - x)f_i + (x - x_i)f_{i+1}}{x_{i+1} - x_i} \quad .} \tag{2.35}$$

Lagrange polynomials

Given $n + 1$ points (x_i, f_i) where $0 \leq i \leq n$, the Lagrange polynomial is an n^{th} degree polynomial that goes exactly through these points;

$$\boxed{P_n(x) = \sum_{i=0}^{n} \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right) f_i \quad ,} \tag{2.36}$$

where in the product, j run over integers from 0 to n but misses out the value i. For example, for the $n = 2$ case

$$P_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f_2 \quad . \tag{2.37}$$

Note that x_i do not need to be equally spaced. Polynomials fit this way can be very wavy.

Bi-linear

This works for a function of two variables $f(x, y)$. Imagine f is known on four points which are the corners of a rectangle in the x–y coordinate system; (x_i, y_k), (x_{i+1}, y_k), (x_i, y_{k+1}) and (x_{i+1}, y_{k+1}). We want f at an arbitrary point (x, y) within this square. Linear interpolation is first applied to f in the x direction, along both the bottom ($y = y_k$) and top ($y = y_{k+1}$) of the square. For instance, along the bottom, equation (2.35) is used with $(x_i, f(x_i, y_k))$ and $(x_{i+1}, f(x_{i+1}, y_k))$ to obtain the intermediate value $f(x, y_k)$. Similarly at the top linear interpolation yields $f(x, y_{k+1})$. Now these two intermediate values are linearly interpolated in the y-direction, using an equation analogous to (2.35). Doing interpolation in y to get the intermediate values and then interpolation in x yields exactly the same value.

Multivariate interpolation is the generalisation to functions with more than one variable; $f(x, y, z, \ldots)$.

SECTION 3
Evaluating Finite Difference Methods

Outline of Section

- Consistency
- Accuracy
- Stability
- Convergence
- Efficiency

The Euler method, seen in section 2.3, is an example of a **finite difference method** (FDM). We will cover a number of more advanced methods for integrating ODEs numerically but before we do that we will look at how to evaluate the properties of any particular method.

3.1. Consistency

This is a check that the finite difference method (e.g. Euler method) reduces to the correct differential equation (e.g. $dy/dt = f(t, y)$) in the limit of vanishingly small step size $h \to 0$. Moreover, a consistency analysis of a finite difference equation reveals the actual differential equation that the finite difference method is effectively solving; the '*modified differential equation*'. The finite difference method actually samples the exact solution of the modified differential equation at the discretized time t_n (or the relevant discretized independent variable η_n). The consistency analysis also tells us the order of the method. This is useful if one is given a finite difference equation, but is not told what its order of accuracy is.

A consistency analysis is carried out by taking the equation for a finite difference method and Taylor expanding all terms about the base point (e.g. η_n). (This is essentially the reverse of the procedure that we used to derive the Euler method in the first place.) As an example, we take the Euler method. Its finite difference equation is

$$y_{n+1} = y_n + f_n\, h \quad , \tag{3.1}$$

where $f_n = f(\eta_n, y_n)$. Next Taylor expand y_{n+1} about the base point y_n. This yields

$$y_n + y'_n\, h + \frac{y''_n}{2}\, h^2 + \frac{y'''_n}{3!}\, h^3 + \ldots = y_n + f_n\, h \quad ,$$

(where $y' = dy/d\eta$ etc.) which can be rearranged into

$$y'_n = f_n - \frac{y''_n}{2}\,h - \frac{y'''_n}{3!}h^2 - \dots \quad . \tag{3.2}$$

Remember that the subscript n notation means evaluation of quantities at η_n. Allowing η_n to become continuous, i.e. $\eta_n \to \eta$ reveals that this is a differential equation

$$\boxed{y' = f(\eta, y) - \frac{y''}{2}\,h - \frac{y'''}{3!}h^2 - \dots} \quad . \tag{3.3}$$

This is the **modified differential equation** (MDE). The discrete points solved by the Euler method (η_n, y_n) lie on the continuous curve that solves the MDE (given the same initial condition of course!). Letting $h \to 0$, equation (3.3) becomes

$$y' = f(\eta, y) \quad , \tag{3.4}$$

which is the intended ODE the Euler method set out to approximate. Thus we say that (3.1) ***is consistent*** with the ODE $y' = f(\eta, y)$. The ***lowest order in h of the correction terms*** in the MDE, e.g. $-y''\,h/2 \sim \mathcal{O}(h)$ in this case, ***is the order of accuracy of the method***. This is the same as the order of the *global error*, discussed in the following section.

Consistency analysis can be carried out with finite difference methods for partial differential equations too (see Section 9).

3.2. Accuracy

To determine the accuracy of a method we want to take the **local error** (truncation error) – the error incurred each step – and use it to estimate the global error (the error at the end of the calculation). Using the Euler method as an example we know that the local truncation error is $\mathcal{O}(h^2)$. [See equation (2.20).] y_{n+1}, the **true** value of the function at η_{n+1}, is related to the numerical approximation of the function $\tilde{y}_{n+1}$ by

$$\tilde{y}_{n+1} \equiv y_n + f_n\,h = y_{n+1} + \mathcal{O}(h^2)\,, \tag{3.5}$$

where we have assumed here that y_n is known exactly. Equation (3.5) shows us the error incurred in performing one step of the Euler method. However to integrate from η_0 to η_{end} would take $\mathcal{O}(1/h)$ steps. If we assume the local error at each of the steps simply adds up, in general, we will have

$$\boxed{\textbf{global error} \approx \text{local error} \times \text{number of steps}\,.} \tag{3.6}$$

In this case

$$\epsilon_{\text{end}} \approx \mathcal{O}(h^2) \times \mathcal{O}(1/h) \sim \mathcal{O}(h)\,. \tag{3.7}$$

In general **a method is called n^{th}-order** if its local error is of $\mathcal{O}(h^{n+1})$, i.e., **its global error is of $\mathcal{O}(h^n)$**.

You might think that all we need to do is reduce the step size h (increase the number of steps) arbitrarily to increase the accuracy arbitrarily. Unfortunately the round–off error places a limit on how accurate any method can be. To see this consider a round-off

error $\mathcal{O}(\mu)$ added to every step in the integration, then the global error after $\mathcal{O}(1/h)$ steps will be

$$\epsilon_{\rm end} \sim \frac{\mu}{h} + h\,. \tag{3.8}$$

Thus there is a minimum error (maximum accuracy) achievable for the Euler method when

$$h \sim \mu^{1/2}\,. \tag{3.9}$$

For double precision on most machines $\mu \sim 10^{-16}$ so the smallest useful step size is $h \sim 10^{-8}$ which will give a global accuracy of $\epsilon_{\rm end} \sim 10^{-8}$.

3.3. Stability

This is a crucial property of a method and describes how an error 'propagates' as the finite difference equation is iterated. If the error increases catastrophically with iteration then the method is unstable. If it doesn't grow or decreases, the method is stable. Imagine that the numerical solution at step n has an error ϵ_n

$$\tilde{y}_n = y_n + \epsilon_n \quad , \tag{3.10}$$

where y_n is the true solution of the ODE at $\eta = \eta_n$ and $\tilde{y}_n$ the numerical approximation. In taking one step of a finite difference method the numerical approximate solution and the error change and satisfy

$$\tilde{y}_{n+1} = y_{n+1} + \epsilon_{n+1} \quad . \tag{3.11}$$

For stability of *any method* we require that the **amplification factor**

$$\boxed{g \equiv \left|\frac{\epsilon_{n+1}}{\epsilon_n}\right| \leq 1 \quad ,} \tag{3.12}$$

otherwise the error in the finite difference method exponentially 'blows up'.

We consider the Euler method as an example. We can write it as

$$\tilde{y}_{n+1} = \tilde{y}_n + f(\eta_n, \tilde{y}_n)\, h \quad . \tag{3.13}$$

Substituting in the true values of the function we have

$$y_{n+1} + \epsilon_{n+1} = y_n + \epsilon_n + f(\eta_n, y_n + \epsilon_n)\, h \quad . \tag{3.14}$$

Stability analysis can only be carried out for linear ODEs. Luckily for non-linear ODEs, i.e., when f is a non-linear function of y, we can still do a stability analysis by assuming that the errors are small $|\epsilon_n|$, $|\epsilon_{n+1}| \ll |y_n|$ and **linearizing $\boldsymbol{f(\eta, y)}$**. To linearize, the function $f(\eta_n, y_n + \epsilon_n)$ is expanded around the point (η_n, y_n) in the variable y. To 1st-order in the expansion parameter ϵ_n the equation (3.14) becomes

$$\begin{aligned} y_{n+1} + \epsilon_{n+1} &= y_n + \epsilon_n + \left[f(\eta_n, y_n) + \left.\frac{\partial f}{\partial y}\right|_n \epsilon_n + \mathcal{O}(\epsilon_n^2) \right] h \quad , \\ &= y_n + f(\eta_n, y_n)h + \epsilon_n \left[1 + \left.\frac{\partial f}{\partial y}\right|_n h \right] + \mathcal{O}(\epsilon_n^2)\,. \end{aligned} \tag{3.15}$$

where $\partial f/\partial y|_n$ means $\partial f/\partial y$ evaluated at the base point $\eta = \eta_n$, $y = y_n$. Then since

$$y_{n+1} = y_n + f(\eta_n, y_n)h + \mathcal{O}(h^2)\,, \tag{3.16}$$

(i.e. Taylor expansion of the true value) and dropping terms of $\mathcal{O}(h^2)$, $\mathcal{O}(\epsilon_n^2)$ and higher we have

$$\boxed{\epsilon_{n+1} \approx \epsilon_n \left(1 + \left.\frac{\partial f}{\partial y}\right|_n h\right)\,,} \tag{3.17}$$

for the Euler method. Inserting $\epsilon_{n+1}/\epsilon_n$ into equation (3.12), and dropping the $|_n$ notation, we can get a condition on the step size h for stability:

$$g = \left|1 + \frac{\partial f}{\partial y}\,h\right| \leq 1 \quad \rightarrow \quad -2 \leq \frac{\partial f}{\partial y}\,h \leq 0\,,$$

then assuming $h > 0$ the Euler method is only stable if

$$\boxed{\frac{\partial f}{\partial y} < 0} \qquad \text{and} \qquad \boxed{h \leq \frac{2}{|\partial f/\partial y|}\,.} \tag{3.18}$$

We say that the Euler method is **conditionally stable** for $\partial f/\partial y < 0$ and **unconditionally unstable** for $\partial f/\partial y > 0$ (i.e. no step size works).

For a non-linear ODE the amplification factor and step size for stability change with y. For a linear ODE, e.g., $dy/d\eta = f(\eta, y) = p(\eta)y + q(\eta)$, we have $\partial f/\partial y = p(\eta)$ and so g and the limiting step size are controlled by $p(\eta)$. (Note that $\partial f/\partial y$ means keeping η fixed.) For constant coefficients, e.g., the decay problem $y' = -\alpha y$ (with α positive), the stability conditions $g = |1 - \alpha\, h|$ and $h \leq 2/\alpha$ do not change over the calculation.

Note that this analysis does not tell us about the global error when the method is stable. Even when $g < 1$ there is still a global error that accumulates with each step. Global error (for a stable method) comes in through the $\mathcal{O}(h^2)$ term dropped in equation (3.16).

3.4. Stability Analysis of General (coupled) Linear Systems

A more general stability analysis can be carried out for m, coupled, linear 1st order ODEs by looking at the general matrix operator form for the method

$$\tilde{\vec{y}}_{n+1} = \mathbf{T}\,\tilde{\vec{y}}_n\,, \tag{3.19}$$

where $\mathbf{T}$ is the update matrix (e.g. $\mathbf{T} = (\mathbf{I} + \mathbf{L}\,h)$ for the Euler method). For a set of inhomogeneous ODEs $\tilde{\vec{y}}_{n+1} = \mathbf{T}\,\tilde{\vec{y}}_n + h\,\vec{q}(\eta_n)$ the term $h\,\vec{q}(\eta_n)$ does not influence numerical stability. Only the homogeneous part of the equation set, i.e., equation (3.19) need be analysed. We can relate the true solution vector $\vec{y}_n = \{y_n^1, y_n^2, \ldots, y_n^m\}$ (where y^i are each of the dependent variables in the coupled ODE set) to the numerical approximation $\tilde{\vec{y}}_n$ in an analogous way to before;

$$\tilde{\vec{y}}_n = \vec{y}_n + \vec{\epsilon}_n$$

where now $\vec{\epsilon}_n = \{\epsilon_n^1, \epsilon_n^2, ..., \epsilon_n^m\}$. Inserting this into the finite difference equation (3.19) yields

$$\vec{y}_{n+1} + \vec{\epsilon}_{n+1} = \mathbf{T}\vec{y}_n + \mathbf{T}\vec{\epsilon}_n\,.$$

We can Taylor expand $\vec{y}_{n+1}$ about $\vec{y}_n$

$$\vec{y}_{n+1} = \vec{y}_n + \vec{f}_n\, h + \mathcal{O}(h^2) = (\mathbf{I} + \mathbf{L}\, h)\, \vec{y}_n + \mathcal{O}(h^2) \approx \mathbf{T}\vec{y}_n \quad ,$$

where $\vec{f}_n = \{f^1(\vec{y}_n), f^2(\vec{y}_n), \ldots, f^m(\vec{y}_n)\}$, i.e., its components are the functions in each ODE $dy^i/d\eta = f^i(\eta, \vec{y})$. This yields a matrix equation for the error propagation;

$$\vec{\epsilon}_{n+1} = \mathbf{T}\, \vec{\epsilon}_n \quad . \tag{3.20}$$

The terms of $\mathcal{O}(h^2)$, discarded in obtaining at (3.20), are the source of global error. However, they are not relevant to the question of stability. It turns out that condition for stability is that the modulus of all the eigenvalues λ^i of the update matrix $\mathbf{T}$ must be less than or equal to unity

$$\boxed{\left|\lambda^i\right| \leq 1 \quad \text{for} \quad i = 1, \ldots, m} \tag{3.21}$$

For real eigenvalues the condition then translates to

$$\lambda^i \leq 1 \quad \text{and} \quad -\lambda^i \leq 1\,, \tag{3.22}$$

To show condition (3.21) it is useful to diagonalise the matrix equation for error propagation so that we are left with m *decoupled* equations which we can deal with individually. Any non-singular $m \times m$ matrix that has m linearly independent eigenvectors can be diagonalised, i.e., written as

$$\boxed{\mathbf{T} = \mathbf{R}\,\mathbf{D}\,\mathbf{R}^{-1} \quad \text{with} \quad \mathbf{D} \equiv \text{diag}(\lambda^1, \lambda^2, ..., \lambda^m)} \quad , \tag{3.23}$$

where λ^i are the eigenvalues of $\mathbf{T}$ and $\mathbf{R}$ is the matrix whose columns are the eigenvectors of $\mathbf{T}$. Substituting (3.23) into (3.20) and operating on both sides with a further $\mathbf{R}^{-1}$ we obtain the diagonal (uncoupled) system

$$\vec{\zeta}_{n+1} = \mathbf{D}\, \vec{\zeta}_n\,, \tag{3.24}$$

where we have defined the linear transformation $\vec{\zeta}_n = \mathbf{R}^{-1}\, \vec{\epsilon}_n$ which rotates the coupled vectors onto an uncoupled basis. This reduces to m uncoupled equations for the rotated errors ζ_n^i

$$\zeta_{n+1}^i = \lambda^i\, \zeta_n^i\,.$$

Since these are uncoupled we can now impose the constraint on their growth separately for each of them, i.e., for all $i = 1, ..., m$ we have the requirement;

$$g^i = \left|\frac{\zeta_{n+1}^i}{\zeta_n^i}\right| = |\lambda_n^i| \leq 1 \quad , \tag{3.25}$$

for a stable method. Since the coupled and uncoupled basis are related by a linear transformation any constraint applied in one basis is equivalent to one applied in the other. That is, the condition (3.25) holds for the original equation (3.20) too.

Non-linear Systems

The above analysis of the eigenvalues of the update matrix can be used for a coupled system of non-linear ODEs, if they are first linearized around the base point $(\eta_n, \vec{y}_n)$. This is necessary to get the update matrix $\mathbf{T}$ that applies at the current step in the iteration.

Considering the Euler method, we have a set of equations

$$\tilde{y}^i_{n+1} = \tilde{y}^i_n + f^i(\eta,\, \tilde{y}^1_n,\, \tilde{y}^2_n,\, \ldots\,,\, \tilde{y}^m_n)\, h \quad . \tag{3.26}$$

We substitute $\tilde{y}^i_n = y^i_n + \epsilon^i_n$ into $f^i(\eta, \tilde{y}^1_n, \ldots)$ and Taylor expand about the true solution to first order in the error, assumed to be small $|\epsilon^i_n| \ll |y^i_n|$;

$$f^i(\eta, y^1_n + \epsilon^1_n,\, y^2_n + \epsilon^2_n,\, \ldots\,,\, y^m_n + \epsilon^m_n) = f^i(\eta, \vec{y}_n) + \sum_{k=1}^{m} \left.\frac{\partial f^i}{\partial (y^k)}\right|_n \epsilon^k_n + \mathcal{O}((\epsilon_n)^2) \quad . \tag{3.27}$$

This is essentially what was done in section 3.3 with equation (3.15). Following the procedure used before we then get

$$\epsilon^i_{n+1} \approx \epsilon^i_n + h \sum_{k=1}^{m} \left.\frac{\partial f^i}{\partial (y^k)}\right|_n \epsilon^k_n \tag{3.28}$$

so that the elements of the update matrix $\mathbf{T}_n$ are

$$\boxed{T_{jk} = \delta_{jk} + h\, \frac{\partial f^j}{\partial (y^k)} \quad ,} \tag{3.29}$$

where δ_{jk} is the Kronecker delta function. (Note that the superscripts in y^j, f^i, etc., denote components and not 'raising to a power'.) The eigenvalues of this update matrix can then be analysed to determine the local stability of the method.

3.5. Convergence

If a finite difference method is both consistent with the ODE (system) being solved and stable, then the approximate solution $\tilde{y}^i_n$ converges to the true solution y_n in the limit as $h \to 0$ (in the absence of round-off error).

3.6. Efficiency

Another consideration when selecting a method is the number of computations required for each step. The Euler method requires 1 functional evaluation for each step. As we will see higher order methods will require more evaluations at each step but will in general be more stable and accurate. A compromise must be reached between efficiency and accuracy and stability for any particular system being solved.

SECTION 4
Methods for Integrating ODE Systems

Outline of Section

- Predictor-Corrector method
- Multi-step methods
- Runge-Kutta methods
- Implicit methods

So far we have only seen one finite difference method (FDM) for integrating ODEs; the Euler method

$$y_{n+1} = y_n + f_n(\eta_n, y_n)\, h \quad .$$

It uses a forward difference scheme (FDS) to approximate y' in $y' = f(\eta, y)$. Its properties are summarised as follows:

- It is a 1st-order method: the global error $\sim \mathcal{O}(h)$.
- The local truncation error (i.e. error per step/iteration) is $\mathcal{O}(h^2)$.
- It is a consistent method.
- It is conditionally stable; $h \leq 2/|\partial f/\partial y|$ for a single ODE.
- It is unstable if $\partial f/\partial y > 0$ (single ODE).
- It is an **explicit** method.
- It is a **single-step** method.

Explicit methods involve evaluating $f(\eta, y)$ using known values of y, e.g., y_n. There are also **implicit** methods which involve evaluating $f(\eta, y)$ using y_{n+1}. **Single-step** methods just need the solution at the current iteration (y_n) to get the next value. **Multi-step** methods require previous values too (e.g. y_{n-1}) in order to get y_{n+1}.

In this section we look at some more explicit FDMs, which are superior to explicit Euler. We also take a brief look at implicit FDMs, concentrating on the implicit Euler method.

Notation: In this section we will drop the '$\sim$' notation for denoting numerical approximate solutions, unless otherwise indicated.

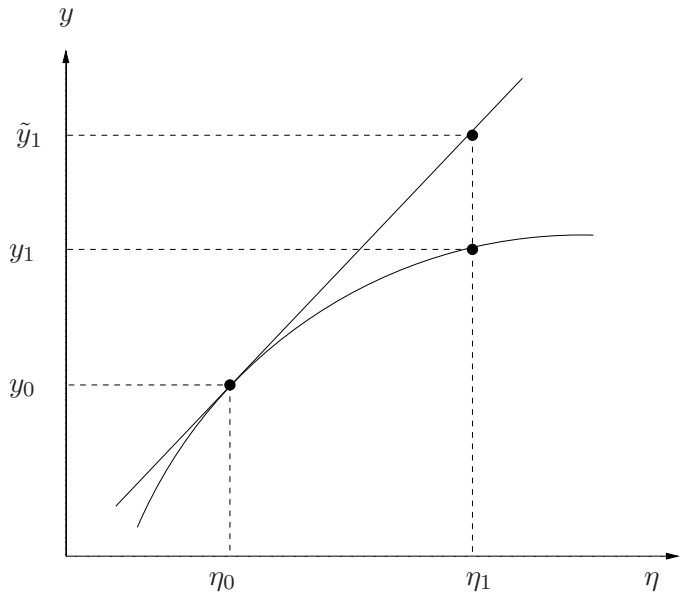

Figure 4.1. The Euler method applied to a non-linear function. The truncation error arises because the gradient is evaluated at the starting point and used to obtain the next point. Here, $\tilde{y}_1$ is the 'approximate' solution according to the Euler method and y_1 (on the curve) is the true solution to the ODE.

4.1. Predictor-Corrector Method

This is classified as an explicit, single-step method. It is a simple improvement to the Euler method. As shown in Fig. 4.1 the Euler method is inaccurate because it uses the gradient evaluated at the initial point to calculate the next point. This only gives a good estimate if the function is linear since the truncation error is quadratic in the step size.

To improve on this we could use the average gradient between the two points

$$y_{n+1} = y_n + \left(\frac{f_n + f_{n+1}}{2}\right) h\,, \tag{4.1}$$

where $f_n \equiv f(\eta_n, y_n)$ etc.

To show that this method is second order accurate (i.e. global error $\sim \mathcal{O}(h^2)$) we can start with the Taylor expansion of y_{n+1}

$$y_{n+1} = y_n + f_n\, h + f'_n \frac{h^2}{2} + f''_n(\xi)\, \frac{h^3}{3!}$$

where as usual $\eta_n < \xi < \eta_n + h$. Above, y_{n+1} is the exact value (at least until the remainder term is dropped). We can replace f'_n in the above Taylor expansion by a forward difference approximation plus a remainder term

$$f'_n = \frac{f_{n+1} - f_n}{h} - f''_n(\zeta)\frac{h}{2}\,,$$

to get

$$y_{n+1} = y_n + f_n\, h + \left(\frac{f_{n+1} - f_n}{h}\right) \frac{h^2}{2} - \left(f''_n(\zeta)\frac{h}{2}\right)\, \frac{h^2}{2} + \frac{1}{3!} f''_n(\xi)\, h^3\,.$$

Expanding out and grouping terms of $\mathcal{O}(h^3)$ we get

$$y_{n+1} = y_n + \frac{f_{n+1} + f_n}{2}\, h + \mathcal{O}(h^3)\,, \tag{4.2}$$

So the local error (truncation) is 3rd order. Since the integration requires $\mathcal{O}(1/h)$ evaluations the **global error is 2nd order** hence Predictor-Corrector is a **2nd-order method**. Using (4.1) will give an approximate solution to the ODE since the remainder terms in the Taylor expansion of y_{n+1} and in using the forward difference approximation of f'_n have been discarded.

The only problem with this method is that we don't have the value y_{n+1} to use in $f_{n+1} = f(\eta_{n+1}, y_{n+1})$ in order to carry out the step. However we can use a first Euler step to *predict* y_{n+1} and use that to calculate f_{n+1} to use in the *corrected* Euler step.

	step 1 (predict):	$y^\star_{n+1} = y_n + f_n\, h \;\;,$
(4.3a)		$f^\star_{n+1} = f(\eta_{n+1}, y^\star_{n+1}) \;\;,$
(4.3b)	step 2 (correct):	$y^{\text{new}}_{n+1} = y_n + \dfrac{f^\star_{n+1} + f_n}{2}\, h \;\;.$

The above boxed equation is the algorithm for the predictor-corrector method, for a single 1st-order ODE.

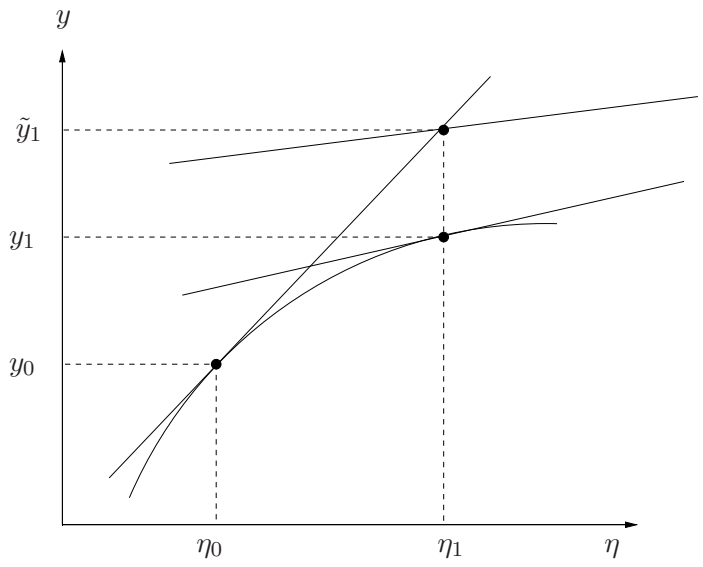

Figure 4.2. (Predictor-corrector) Ideally we would like to use the gradient at y_{n+1} to get the average gradient in the interval. However we do not have the true value y_{n+1} so we use the gradient calculated with the Euler estimate $\tilde{y}_{n+1}$. (Nb: $n = 0$ is used in the plot.)

The gradient calculated with the predicted point will not, in general, be exactly the correct value since it depends on both η and y but it will still be more accurate than the Euler method (see Fig. 4.2). The predictor-corrector method is $\mathcal{O}(h^2)$ accurate but involves two evaluations per step which is less efficient. It is conditionally stable for decaying solutions ($y' = -\alpha y$), but unstable for pure oscillating solutions ($y'' = -\omega^2 y$).

Coupled equations

For a set of m coupled 1st-order ODEs, step 1 should first be applied to each of the m ODEs to get the predicted value $(y^\star_{n+1})^i$ for each dependent variable y^i. Then $(f^\star_{n+1})^i = f^i(\eta_{n+1},\, \vec{y}^\star_{n+1})$ can be evaluated for each ODE. Then step 2 (corrected Euler) can be applied to each ODE. In the compact vector notation, this is

$$\text{step 1 (predict):} \qquad \vec{y}^\star_{n+1} = \vec{y}_n + \vec{f}_n\, h \ , \tag{4.4a}$$

$$\vec{f}^\star_{n+1} = \vec{f}(\eta_{n+1}, \vec{y}^\star_{n+1}) \ , \tag{4.4b}$$

$$\text{step 2 (correct):} \qquad \vec{y}^{\,\text{new}}_{n+1} = \vec{y}_n + \frac{\vec{f}^\star_{n+1} + \vec{f}_n}{2}\, h \ . \tag{4.4c}$$

4.2. Multi-Step (Leapfrog) Method

We have seen how the use of a central gradient between two points is more accurate ($\mathcal{O}(h^2)$) than using the gradient at the first point ($\mathcal{O}(h)$). In the previous section we predicted the next point to take an average of the gradient between y_n and y_{n+1}. However since we have presumably evaluated values before y_n in previous steps we could use those to get an update. This requires storing a number of previous points.

The simplest of these methods is the Leapfrog method which uses the y_{n-1} value

$$\boxed{y_{n+1} = y_{n-1} + 2f_n\, h \ ,} \tag{4.5}$$

and requires the storage of one previous step. In this method the point (η_n, y_n) – where the gradient is used – is the midpoint between y_{n-1} and y_{n+1}. The leapfrog algorithm therefore essentially uses a central difference scheme to approximate the derivative y' in the ODE. The leapfrog method is an explicit FDM. The Leapfrog ($m = 1$ order multi-step) method is **$\mathcal{O}(h^2)$ accurate** and only requires one evaluation per step. This makes it more efficient than the Predictor-Corrector method. The method is stable for oscillating and growing solutions but is unstable for decaying solutions.

General multi-step

This kind of multi-step method can use any number of previous values, say 'm'. The Leapfrog is an $m = 1$ order multi-step method. In general, an m^{th}-order multi-step method is an **$(m+1)^{\text{th}}$ order accurate method**.

One way to get an m^{th}-order multi-step method is to fit $P_{m+1}(\eta)$, an $(m+1)^{\text{th}}$-degree Lagrange polynomial, through the $m+2$ points y_{n+1}, y_n, y_{n-1}, … , y_{n-m} (i.e. including the point to be determined) and then approximate y' in the ODE by P'_{m+1} (for which an analytic expression can easily be obtained, once the coefficients of the interpolating polynomial have been found). Another viable way is to fit $P_m(\eta)$ to $f(\eta, y)$ through the $m+1$ points (η_n, f_n), (η_{n-1}, f_{n-1}), … , (η_{n-m}, f_{n-m}) and then analytically integrate this interpolated $f(\eta, y)$ from η_n to η_{n+1} thus advancing the solution to y_{n+1}. This yields the 'Adam-Bashforth' family of methods.

However the drawback of multi-step methods is that $m+1$ points are needed. So to apply a multi-step method from the outset, values before $\eta = \eta_0$ have to be guessed. If the guess is not a good one then the method can suffer from a initial value error.

Starting off

Typically, what is done is to use a single-step method for the first iteration(s). For instance to start off the leapfrog method, an Euler step can be used to get y_1 from the initial condition y_0. Then there are enough points to continue with the leapfrog scheme:

$$\begin{aligned} y_1 &= y_0 + f_0\,h\,, \\ y_2 &= y_0 + 2f_1\,h\,, \\ y_3 &= y_1 + 2f_2\,h\,, \\ &\ldots \\ y_{n+1} &= y_{n-1} + 2f_n\,h\,. \end{aligned} \tag{4.6}$$

To improve the accuracy of the starting step, a better single-step method could be used and/or smaller steps can be used (e.g. use k Euler steps with $h \to h/k$ to get y_1 for starting leapfrog).

Variable step size h

With single-step methods, it is easy to vary h during the calculation to, e.g., keep the error with a specified bounds. ('Adaptive step-size' algorithms can be found in any good numerical methods book; we won't cover them here.) Another disadvantage of multi-step methods, in comparison to single-step methods, is that varying h is more difficult (requiring back-calculating y_{n-1}, etc., using a good single-step method).

4.3. Runge-Kutta Methods

The Runge-Kutta methods are a family of explicit, single-step methods using more than one term in the Taylor series expansion of y_{n+1}. The method generalises the idea, employed by the Predictor-Corrector scheme, of using a weighted average of gradients, to using m estimates of the gradient calculated at various points in the interval $\eta_n \leq \eta \leq \eta_{n+1}$ to determine the change in y.

In general the **RKm** method equates to an m^{th}-order Taylor expansion, and is an **m^{th}-order finite difference method**. Thus the local error (truncation) is of $\mathcal{O}(h^{m+1})$ and the global accuracy of the method is $\mathcal{O}(h^m)$. A number of weighting schemes for the averaging of the gradients can be used for each RKm.

RK2

The RK2 class of methods includes the Predictor-Corrector method discussed in section 4.1. The general RK2 scheme is;

$$y_{n+1} = y_n + a\,k_1 + b\,k_2 \quad , \tag{4.7a}$$

$$k_1 = h\,f(\eta_n, y_n) \quad , \tag{4.7b}$$

$$k_2 = h\,f(\eta_n + \alpha\,h, y_n + \beta\,k_1) \quad . \tag{4.7c}$$

Here a, b, α, and β are coefficients. Different choices for the coefficients give different methods

$$
\begin{aligned}
&a = 0\,, \quad b = 1\,, \quad \alpha = \tfrac{1}{2}\,, \quad \beta = \tfrac{1}{2} && \text{:Single mid-point}\\
&a = \tfrac{1}{2}\,, \quad b = \tfrac{1}{2}\,, \quad \alpha = 1\,, \quad \beta = 1 && \text{:Predictor-Corrector}\\
&a = \tfrac{1}{3}\,, \quad b = \tfrac{2}{3}\,, \quad \alpha = \tfrac{3}{4}\,, \quad \beta = \tfrac{3}{4} && \text{:Smallest error RK2}
\end{aligned}
\tag{4.8}
$$

The RK2 method is illustrated in Fig. 4.3. It uses two stages of gradient estimation. At each stage, the gradient is used to estimate the change in y going from η_n to η_{n+1}. The first estimate is an Euler step yielding an estimated shift in y of $k_1 \equiv y^{(1)}_{n+1} - y_n$. In the second stage, a more accurate value of the gradient is estimated using the function f at a point shifted by an amount $\alpha\, h$ in η and by an amount $\beta\, k_1$ in y. This yields a better estimate in the shift in y, i.e., $k_2 = y^{(2)}_{n+1} - y_n$. The two shifts are then averaged with weights a and b.

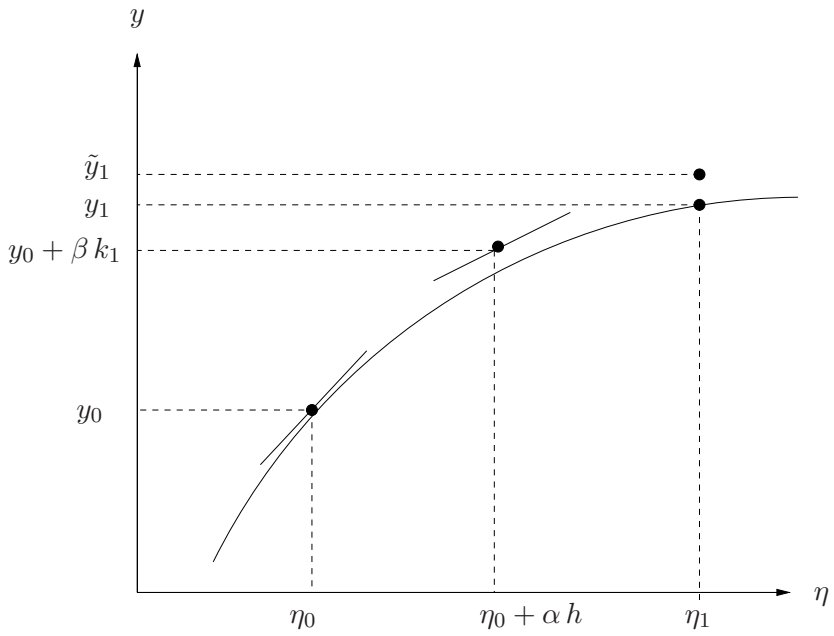

Figure 4.3. The RK2 method. A second gradient is calculated at a shifted position and averaged with the gradient at the starting position to obtain $\tilde{y}_1$, the next value of y. Comparing to fig. 4.1 you can see that $\tilde{y}_1$ is more accurate for RK2 compared to Euler.

Coupled ODEs

For coupled ODEs, the first estimate stage must be carried out for all the dependent variables (e.g. u, v, w in $\vec{y} = \{u, v, w\}$) before moving onto the 2nd estimation stage.

The RK2 scheme is then;

$$\vec{y}_{n+1} = \vec{y}_n + a\,\vec{k}_1 + b\,\vec{k}_2 \quad , \tag{4.9a}$$

$$\vec{k}_1 = h\,\vec{f}(\eta_n, \vec{y}_n) \quad , \tag{4.9b}$$

$$\vec{k}_2 = h\,\vec{f}(\eta_n + \alpha\,h, \vec{y}_n + \beta\,\vec{k}_1) \quad . \tag{4.9c}$$

RK4

This uses 4 stages of gradient estimation to calculate an average one and is 4^{th}-order accurate.

$$\boxed{\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}\,(k_1 + 2\,k_2 + 2\,k_3 + k_4) \quad , && (4.10a)\\ k_1 &= h\,f(\eta_n, y_n) \quad , && (4.10b)\\ k_2 &= h\,f(\eta_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_1) \quad , && (4.10c)\\ k_3 &= h\,f(\eta_n + \frac{1}{2}\,h, y_n + \frac{1}{2}\,k_2) \quad , && (4.10d)\\ k_4 &= h\,f(\eta_n + h, y_n + k_3) \quad . && (4.10e) \end{aligned}}$$

Again the first stage is an Euler estimate, as in RK2. The 2nd and 3rd stages estimate the gradient using a fraction $\frac{1}{2}$ of the preceding shifts, k_1 and k_2, respectively. The 4th stage estimates the gradient using the gradient function f evaluated at the best estimate yet of the true value of y_{n+1}, i.e., $y^{(3)}_{n+1} = y_n + k_3$ and η_{n+1}. The values of the weights and shift coefficients are careful chosen to minimise the error.

The advantage of the RK4 method is that, although it requires four evaluations per step, it can take much larger values of h and is therefore more efficient than the Euler and Predictor-Corrector methods. Even higher order RK can be used but these require more (e.g. 11 for RK6) evaluations per step and are therefore slower. Only for $m < 4$ are $\leq m$ evaluations required per step. One practical advantage of RK methods over multi-step methods is that we don't need to store any previous steps. This makes coding it a little easier.

4.4. Implicit Methods

Implicit methods for solving an ODE require evaluation of $f(\eta, y)$ using the yet unknown value y_{n+1}. The simplest is implicit Euler:

$$\boxed{y_{n+1} = y_n + f(\eta_{n+1}, y_{n+1})\,h \quad .} \tag{4.11}$$

Implicit Euler is still a 1^{st}-order method, with global accuracy $\mathcal{O}(h)$, just like its explicit cousin. For a linear function $f(\eta, y) = p(\eta)y + q(\eta)$ equation (4.11) can easily be rearranged to give $y_{n+1} = g(\eta_{n+1}, y_n)$ (where g is a new function) which can easily be solved. For a non-linear function, (4.11) is a non-linear algebraic equation in y_{n+1},

that needs to be solved using something like the bisection, Newton or secant method (see section 2.5).

Stability

What is the advantage then? The advantage of implicit Euler over explicit Euler is that it is **unconditionally stable**. There is no critical step size h, above which numerical instability occurs. Of course, using a large h in implicit Euler will give a very inaccurate solution. The stability analysis carried out in section 3.3 (for explicit Euler) can be applied to implicit Euler, in much the same way, yielding

$$\frac{\epsilon_{n+1}}{\epsilon_n} \approx \left(1 - \frac{\partial f}{\partial y}\, h\right)^{-1} \tag{4.12}$$

so that for decaying problems ($\partial f/\partial y < 0$) we have $g \leq 1$ for ***any*** (positive) h. What happens to y_{n+1} for large h? It simply tends to zero; the same behaviour as the exact solution. Inspection of equation (4.11) for the linear decay problem ($f = -\beta\, y$) shows that $y_{n+1} = y_n/(1 + \beta\, h) \to 0$ as $h \to \infty$.

Just like we have seen more advanced explicit methods (than Euler), more advanced implicit methods exist and have superior stability characteristics to explicit methods.

Coupled ODEs

Implicit Euler works for coupled, linear, 1st-order ODEs. One has

$$\begin{aligned} & & \vec{y}_{n+1} &= \vec{y}_n + h\mathbf{L} \cdot \vec{y}_{n+1} \\ \therefore & & (\mathbf{I} - h\mathbf{L}) \cdot \vec{y}_{n+1} &= \vec{y}_n \\ & \therefore & \vec{y}_{n+1} &= (\mathbf{I} - h\mathbf{L})^{-1} \cdot \vec{y}_n = \bar{\mathbf{T}} \cdot \vec{y}_n \end{aligned} \tag{4.13}$$

where $\mathbf{L}$ is the same matrix operator as for explicit Euler. $\bar{\mathbf{T}}$ is the update matrix for implicit Euler and is the inverse of the matrix $(\mathbf{I} - h\mathbf{L})$. Hence implicit Euler works if $(\mathbf{I} - h\mathbf{L})$ is a non-singular matrix so that $\bar{\mathbf{T}}$ exists and can be found.

With non-linear coupled ODEs, things are not so easy. In principle one needs to find the solution of a set of (coupled) non-linear algebraic equations. This is at best tricky and at worst insoluble. A way out is to linearize the functions f^i in each ODE about the known 'point' $(\eta_n, \vec{y}_n)$. But then the update matrix $\bar{\mathbf{T}}$ has to be calculated at each step, since the matrix $\mathbf{L}$ obtained by linearisation depends on $\vec{y}_n$. Thus implicit methods are generally less efficient than explicit methods.