Chapter 2

Introduction to Numerical Calculations

Outline of Section

- Nature of errors
- Natural units
- Solving non-linear algebraic equations

2.1 Numerical Accuracy and Errors

Review of Taylor series

In this course, we will 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

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

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 $x \leq \xi \leq x+h$ and $y \leq \eta \leq y+k$. 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

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

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 $-\mathbf{1.665e{-}16}$ 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

$$\begin{array}{rcl} \text{binary representation} & & \text{decimal representation} \\ 0 & = & 0 \\ 1 & = & 1 \\ 10 & = & 2 \\ 11 & = & 3 \\ 100 & = & 4 \\ 101 & = & 5 \\ 110 & = & 6 \\ 111 & = & 7 \\ 1000 & = & 8 \\ \text{etc...} & & \end{array} \tag{2.7}$$

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}

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

(2.8)

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.

It's important that you know about how floating-point numbers are represented on computers, how computers operate on them to perform subtraction, addition, multiplication and division, and what the consequences are for usable accuracy, rounding and truncation. The best place to read more about that is in Golberg, 1992 available on Blackboard.

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

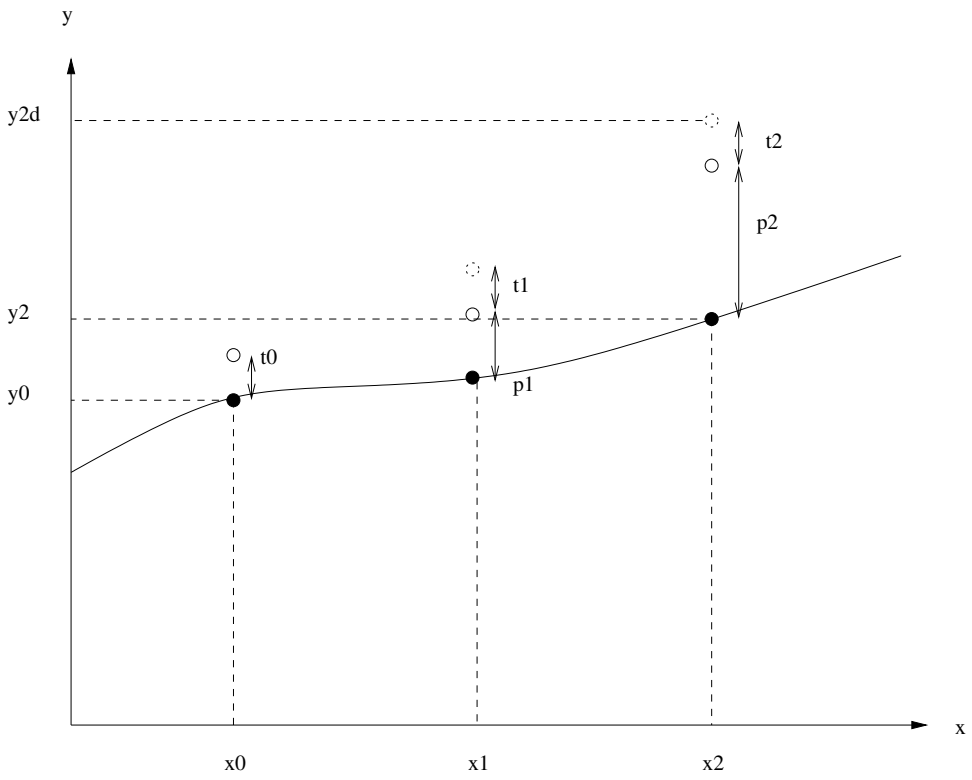

Figure 2.1: Illustration of truncation and propagating errors incurred in solving an ODE $dy/dx = f(y, x)$. The solid line depicts the exact solution. Dashed open circles depict the numerical solution. The calculation starts at (x_0, y_0) (off the plot) where there is no error. $\tilde{y}_3$ is the numerical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

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 ordinary differential equation (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 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 dimensionful system for exponential decay with a source

$$\frac{dy}{dt} + \alpha\, y \;=\; g \tag{2.9}$$

where α is a decay rate and g a constant growth/source term. Assuming the dependent variable

has SI dimensions $[y] = \mathrm{m}$, then $[\alpha] = \mathrm{s}^{-1}$ and $[g] = \mathrm{m/s}$. We can then define a scaled time variable $\hat{t} = \alpha\, t$ such that [1]

$$\frac{d}{dt} = \frac{d\hat{t}}{dt}\frac{d}{d\hat{t}} = \alpha\frac{d}{d\hat{t}} \tag{2.10}$$

and the equation becomes

$$\frac{dy}{d\hat{t}} + y \;=\; \frac{g}{\alpha} \equiv G, \tag{2.11}$$

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

$$\frac{d\hat{y}}{d\hat{t}} + \hat{y} \;=\; 1. \tag{2.12}$$

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 $[g/\alpha]$.

2.3 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 $\sin\!\left(x^2\right) - 1/(x + a) = 0$. 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)

[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).

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

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.} \tag{2.13}$$

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,

$$\epsilon_{i+1} = \text{const} \times (\epsilon_i)^2. \tag{2.14}$$

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)

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}. \tag{2.15}$$

Inserting into Newton's method gives

$$\boxed{x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}.} \tag{2.16}$$

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

Brent's method

This is the Rolls Royce of 1D root-finding. It combines bisection with inverse quadratic interpolation *and* the secant method, plus careful bracketing and error tracking — to give what is basically the last algorithm you will ever need for solving equations in one variable. It has the downside of being quite fiddly to code though, as it flips back and forth between the bisection, inverse quadratic and secant methods according to a series of different criteria, depending on how it is tracking at the time. You can read a good account of it, and see some example code, in Numerical Recipes.