Notes 8: Solving ODEs and Fourier transforms

8.1 Ordinary differential equations and dynamical systems

Differential equations are one of the basic tools of mathematical modelling and no discussion of computational methods would be complete without a discussion of methods for their numerical solution.. An ordinary differential equation (ODE) is an equation involving one or more derivatives of a function of a single variable. The word "ordinary" serves to distinguish it from a partial differential equation (PDE) which involves one or more partial derivatives of a function of several variables. Let u be a real–valued function of a single real variable, t, and F be a real–valued function of t, $u(t)$ and its first $m-1$ derivatives. The equation

$$\frac{d^m u}{dt^m}(t) = F\left(t, u(t), \frac{du}{dt}(t), \ldots, \frac{d^{m-1}u}{dt^{m-1}}(t)\right) \tag{8.1}$$

is an ordinary differential equation of order m. Equations for which F is has no explicit dependence on t are referred to as autonomous. The general solution of an $m^{\rm th}$ order ODE involves m constants of integration. We frequently encounter three types of problems involving differential equations:

- **Initial Value Problems (IVPs)**
  The value of $u(t)$ and a sufficient number of derivatives are specified at an initial time, $t = 0$ and we require the values of $u(t)$ for $t > 0$.
- **Boundary Value Problems (BVPs)**
  Boundary value problems arise when information on the behaviour of the solution is specified at two different points, x_1 and x_2, and we require the solution, $u(x)$, on the interval $[x_1, x_2]$. Choosing to call the dependent variable x instead of t hints at the fact that boundary value problems frequently arise in modelling spatial behaviour rather than temporal behaviour.
- **Eigenvalue Problems (EVPs)**
  Eigenvalue problems are boundary value problems involving a free parameter, λ. The objective is to find the value(s) of λ, the eigenvalue(s), for which the problem has a solution and to determine the corresponding eigenfunction(s), $u_\lambda(x)$.

We will focus exclusively on IVPs here but much of what we learn is relevant for BVPs and EVPs.

Systems of coupled ODEs arise very often. They can arise as a direct output of the modelling or as a result of discretising the spatial part of a PDE. Many numerical approaches to solving PDEs involve approximating the PDE by a (usually very large) system of ODEs. In practice, we only need to deal with systems of ODEs of first order. This is because an ODE of order m can be rewritten as a system of first order ODEs of dimension m. To do this for the $m^{\rm th}$ order ODE Eq.(8.1), define a vector $\mathbf{u}(t)$ in $\mathbb{R}^m$ whose components [1], $u^{(i)}(t)$, consist of the function $u(t)$ and its first $m-1$ derivatives:

$$\mathbf{u}(t) = \left(u^{(0)}(t), \ldots u^{(m-1)}(t)\right) = \left(u(t), \frac{du}{dt}(t), \ldots, \frac{d^{m-1}u}{dt^{m-1}}(t)\right).$$

From this point on, we will not write the explicit t-dependence of $\mathbf{u}(t)$ and its components unless it is necessary for clarity. We define another vector, $\mathbf{F} \in \mathbb{R}^m$, whose components will give the right hand sides:

$$\mathbf{F}(t, \mathbf{u}) = \left(u^{(1)}, \ldots, u^{(m-1)}, F(t, \mathbf{u})\right).$$

By construction, Eq.(8.1) is equivalent to the first order system of ODEs

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}\left(t, \mathbf{u}\right). \tag{8.2}$$

[1] The clumsy superscript notation, $u^{(i)}$ for the components of $\mathbf{u}$ is adopted to accommodate the subsequent use of the notation u_i to denote the value of a function $u(t)$ at a discrete time point, t_i.

The initial conditions given for Eq. (8.1) can be assembled together to provide an initial condition $\mathbf{u}(0) = \mathbf{U}$. We can also do away with the distinction between autonomous and non-autonomous equations be introducing an extra component, $u^{(m)}$, of $\mathbf{u}$ which satisfies the trivial ODE

$$\frac{du^{(m)}}{dt} = 1 \qquad \text{with } u^{(m)}(0) = 0. \tag{8.3}$$

For this reason, we will focus primarily on developing numerical algorithms for autonomous first order systems of equations in dimension m

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}\left(\mathbf{u}\right) \qquad \text{with } \mathbf{u}(0) = \mathbf{U}. \tag{8.4}$$

8.2 Timestepping and simple Euler methods

8.2.1 Timestepping and the Forward Euler Method

In the computer, we approximate $\mathbf{u}$ by its values at a finite number of points. The process of replacing a function by an array of values at particular points is called discretisation. We replace $\mathbf{u}(t)$ with $t \in [t_1, t_2]$ by a list of values, $\{\mathbf{u}_i \ : \ i = 0 \dots N\}$ where $\mathbf{u}_i = \mathbf{u}(t_i)$, $t_i = t_1 + ih$ and $h = (t_2 - t_1)/N$. There is no reason why the discretisation should be on a set of uniformly spaced points but we will initially consider this case. We have already seen in Sec. 3.2.1 how knowing the values of a function at a stencil of discrete points allows us to approximate the derivative of the function. Furthermore we knew the error in the approximation using Taylor's Theorem. We can turn this reasoning around: if we know $\mathbf{u}_i$ and $\frac{d\mathbf{u}}{dt}(t_i)$ then we can approximate $\mathbf{u}_{i+1}$ (of course since $\mathbf{u}(t)$ is now a vector-valued function we must use the multivariate version of Taylor's theorem):

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\frac{d\mathbf{u}}{dt}(t_i) + O(h^2). \tag{8.5}$$

For the system of ODEs (8.4) this gives:

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\mathbf{F}_i + O(h^2), \tag{8.6}$$

where we have adopted the obvious notation $\mathbf{F}_i = \mathbf{F}(\mathbf{u}(t_i))$. Starting from the initial condition for Eq. (8.4) and knowledge of $\mathbf{F}(\mathbf{u})$ we obtain an approximate value for $\mathbf{u}$ a time h later. We can then iterate the process to obtain an approximation to the values of $\mathbf{u}$ at all subsequent times, t_i. While this iteration might be possible to carry out analytically in some simple cases (for which we are likely to be able to solve the original equation anyway), in general it is ideal for computer implementation. This iteration process is called time-stepping. Eq. (8.6) provides the simplest possible time-stepping method. It is called the Forward Euler Method.

8.2.2 Stepwise vs Global Error

In Eq. (8.6), we refer to $O(h^2)$ as the stepwise error. This is distinct from the global error, which is the total error which occurs in the approximation of $\mathbf{u}(t_2)$ if we integrate the equation from t_1 to t_2 using a particular timestepping algorithm. If we divide the time domain into N intervals of length $h = (t_2 - t_1)/N$ then $N = (t_2 - t_1)/h$. If we make a stepwise error of $O(h^n)$ in our integration routine, then the global error therefore $O((t_2 - t_1)h^{n-1})$. Hence the global error for the Forward Euler Method os $O(h)$. This is very poor accuracy. This is one reason why the Forward Euler Method is almost never used in practice.

8.2.3 Backward Euler Method - implicit vs explicit timestepping

We could equally well have used the backward difference formula, Eq. (3.9), to derive a time-stepping algorithm in which case we would have found

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\mathbf{F}_{i+1} + O(h^2). \tag{8.7}$$

Now, $\mathbf{u}_{i+1}$ (the quantity we are trying to find) enters on both sides of the equation. This means we are not, in general in a position to write down an explicit formula for $\mathbf{u}_{i+1}$ in terms of $\mathbf{u}_i$ as we did for the Forward Euler Method, Eq. (8.6). Rather we are required to solve a (generally nonlinear) system of equations to find the value of $\mathbf{u}$ at the next timestep. For non-trivial $\mathbf{F}$ this may be quite hard since, as we learned in Sec. 4.2, root finding in higher dimensions can be a difficult task.

The examples of the Forward and Backward Euler methods illustrates a very important distinction between different timestepping algorithms: explicit vs implicit. Forward Euler is an explicit method. Backward Euler is implicit. Although they both have the same degree of accuracy - a stepwise error of $O(h^2)$ - the implicit version of the algorithm typically requires a lot more work per timestep. However it has superior stability properties.

8.3 Predictor-Corrector methods

8.3.1 Implicit Trapezoidal method

We arrived at the simple Euler methods by approximating $\mathbf{u}$ by its Taylor series. A complementary way to think about it is to begin from the formal solution of Eq. (8.4):

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \int_{t_i}^{t_i+h} \mathbf{F}(\mathbf{u}(\tau))\, d\tau, \tag{8.8}$$

and think of how to approximate the integral,

$$I = \int_{t_i}^{t_i+h} \mathbf{F}(\mathbf{u}(\tau))\, d\tau. \tag{8.9}$$

The simplest possibility is to use the rectangular approximations familiar from the definition of the Riemann integral. We can use either a left or right Riemann approximation:

$$I \approx h\mathbf{F}(\mathbf{u}(t_i)) \tag{8.10}$$
$$I \approx h\mathbf{F}(\mathbf{u}(t_{i+1})). \tag{8.11}$$

These approximations obviously give us back the Forward and Backward Euler Methods which we have already seen. A better approximation would be to use the Trapezoidal Rule, Eq. (3.12) :

$$I \approx \frac{1}{2}h\left[\mathbf{F}(\mathbf{u}(t_i)) + \mathbf{F}(\mathbf{u}(t_{i+1}))\right].$$

Eq. (8.8) then gives the timestepping rule

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}(\mathbf{F}_i + \mathbf{F}_{i+1}). \tag{8.12}$$

This is known as the Implicit Trapezoidal Method. The stepwise error in this approximation is $O(h^3)$. To see this, we Taylor expand the terms sitting at later times in Eq. (8.12) and compare the resulting expansion to the true Taylor expansion. It is worth writing this example out in some detail since it is a standard way of deriving the stepwise error for any timestepping algorithm. For simplicity, let us assume that we have a scalar equation,

$$\frac{du}{dt} = F(u). \tag{8.13}$$

The extension of the analysis to the vector case is straight-forward but indicially messy. The true solution at time t_{i+1} up to order h^3 is, from Taylor's Theorem and the Chain Rule:

$$\begin{aligned} u_{i+1} &= u_i + h\frac{du}{dt}(t_i) + \frac{h^2}{2}\frac{d^2u}{dt^2}(t_i) + O(h^3) \\ &= u_i + hF_i + \frac{1}{2}h^2 F_i\, F_i' + O(h^3) \end{aligned}$$

Note that we have used Eq. (8.13) and the chain rule to replace the second derivative of u wrt t at time t_i by $F_i\, F_i'$. Make sure you understand how this works. Let us denote our approximate solution by $\widetilde{u}(t)$. From Eq. (8.12)

$$\widetilde{u}_{i+1} = u_i + \frac{h}{2}\left[F(u_i) + F(u(t_{i+1}))\right]$$

We can write

$$\begin{aligned} F(u(t_{i+1})) &= F(u(t_i)) + h\frac{dF}{dt}(u(t_i)) + O(h^2) \\ &= F_i + hF_i'F_i + O(h^2). \end{aligned}$$

Substituing this back gives

$$\widetilde{u}_{i+1} = u_i + hF_i + \frac{1}{2}h^2 F_i\,F_i' + O(h^3).$$

We now see that the difference between the true value of u_{i+1} and the approximate value given by Eq. (8.12) is $\widetilde{u}_{i+1} - u_{i+1} = O(h^3)$. Hence the Implicit Trapezoidal Method has an $O(h^3)$ stepwise error. Strictly speaking we have only shown that the stepwise error is at most $O(h^3)$. We have not computed the $O(h^3)$ terms explicitly to show that they do not in general cancel as the $O(h^2)$ ones do. Take it on faith that they do not (or do the calculation yourself in an idle moment).

8.3.2 Improved Euler method

The principal drawback of the Implicit Trapezoidal Method is that it is implicit and hence computationally expensive and tricky to code (but not for us now that we know how to find roots of nonlinear equations!). Can we get higher order accuracy with an explicit scheme? A way to get around the necessity of solving implicit equations is try to "guess" the value of $\mathbf{u}_{i+1}$ and hence estimate the value of $\mathbf{u}_{i+1}$ to go into the RHS of Eq. (8.12). How do we "guess"? One way is to use a less accurate explicit method to predict $\mathbf{u}_{i+1}$ and then use a higher order method such as Eq. (8.12) to correct this prediction. Such an approach is called a Predictor–Corrector Method. Here is the simplest example:

- Step 1
  Make a prediction, $\mathbf{u}^*_{i+1}$ for the value of $\mathbf{u}_{i+1}$ using the Forward Euler Method:

  $$\mathbf{u}^*_{i+1} = \mathbf{u}_i + h\mathbf{F}_i,$$

  and calculate

  $$\mathbf{F}^*_{i+1} = \mathbf{F}(\mathbf{u}^*_{i+1}).$$

- Step 2
  Use the Trapezoidal Method to correct this guess:

  $$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}\left[\mathbf{F}_i + \mathbf{F}^*_{i+1}\right]. \tag{8.14}$$

Eq. (8.14) is called the Improved Euler Method. It is explicit and has a stepwise error of $O(h^3)$. This can be shown using an analysis similar to that done above for the Implicit Trapezoidal Method. The price to be paid is that we now have to evaluate $\mathbf{F}$ twice per timestep. If $\mathbf{F}$ is a very complicated function, this might be worth considering but typilcally the improved accuracy more than compensates for the increased number of function evaluations.

There are two common approaches to making further improvements to the error:

1. Use more points to better approximate the integral, Eq. (8.9). This leads to class of algorithms known as multistep methods. These have the disadvantage of needing to remember multiple previous values (a problem at $t = 0$!) and will not be discussed here although you can read about them here [1] or [2, Chap. 17].
2. Use predictors of the solution at several points in the interval $[t_i, t_{i+1}]$ and combine them in clever ways to cancel errors. This approach leads to a class of algorithms known as Runge-Kutta methods.

8.4 Using adaptive timestepping to deal with multiple time scales and singularities

Up until now we have characterised timestepping algorithms as $h \to 0$. In practice we need to operate at a finite value of h. How do we choose it? Ideally we would like to choose the timestep such that the error per timestep is less than some threshold, ϵ. We measure the error by comparing

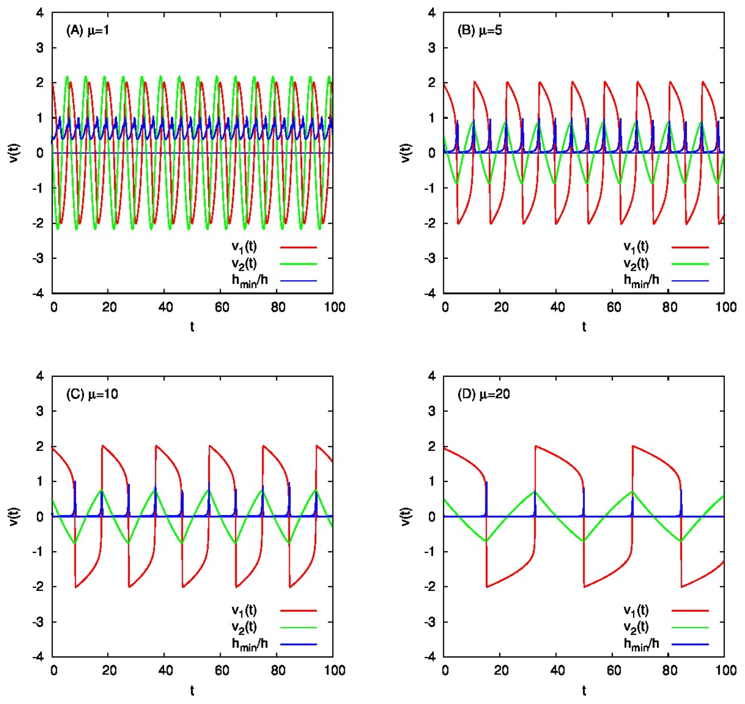

Figure 8.1: Solutions of Eq. (8.15) for several different values of μ obtained using the forward Euler Method with adaptive timestepping. Also shown is how the timestep varies relative to its global minimum value as the solution evolves.

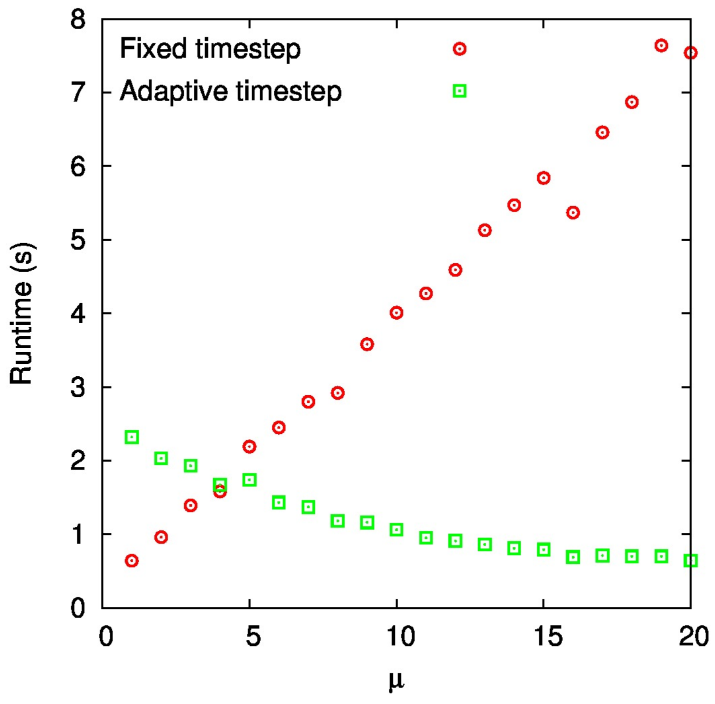

Figure 8.2: Comparison of the performance of the simple Euler method with adaptive and fixed timestepping strategies applied to Eq. (8.15) for several different values of μ.

the numerical solution at a grid point, $\widetilde{\mathbf{u}}_i$, to the exact solution, $\mathbf{u}(t_i)$, assuming it is known. Two criteria are commonly used:

$$\begin{aligned} E_a(h) &= |\widetilde{\mathbf{u}}_i - \mathbf{u}_i| \leq \epsilon \quad \text{Absolute error threshold,} \\ E_r(h) &= \tfrac{|\widetilde{\mathbf{u}}_i - \mathbf{u}_i|}{|\mathbf{u}_i|} \leq \epsilon \quad \text{Relative error threshold.} \end{aligned}$$

Intuitively (and mathematically from Taylor's Theorem) the error is largest when the solution is rapidly varying. Often the solution does not vary rapidly at all times. By setting the timestep in this situation so that the error threshold is satisfied during the intervals of rapid variation, we end up working very inefficiently during the intervals of slower variation where a much larger timestep would have easily satisfied the error threshold. Here is a two–dimensional example which you already know: a relaxation oscillator when $\mu \gg 1$:

$$\begin{aligned} \frac{dx}{dt} &= \mu(y - (\frac{1}{3}x^3 - x)) \\ \frac{dy}{dt} &= -\frac{1}{\mu}x. \end{aligned} \tag{8.15}$$

The solution of this system has two widely separated timescales - jumps which proceed on a time of $O(\frac{1}{\mu})$ (very fast when $\mu \gg 1$) and crawls which proceed on a timescale of μ (very fast when $\mu \gg 1$). To integrate Eqs. (8.15) efficiently for a given error threshold, we need to take small steps during the jumps and large ones during the crawls. This process is called adaptive timestepping.

The problem, of course, is that we don't know the local error for a given timestep since we do not know the exact solution. So how do we adjust the timstep if we do not know the error? A common approach is to use trial steps: at time i we calculate one trial step starting from the current solution, $\mathbf{u}_i$ using the current value of h and obtain an estimate of the solution at the next time which we shall call $\mathbf{u}^{\mathrm{B}}_{i+1}$. We then calculate a second trial step starting from $\mathbf{u}_i$ and taking two steps, each of length $h/2$, to obtain a second estimate of the solution at the next time which we shall call $\mathbf{u}^{\mathrm{S}}_{i+1}$. We can then estimate the local error as

$$\Delta = \left|\mathbf{u}^{\mathrm{B}}_{i+1} - \mathbf{u}^{\mathrm{S}}_{i+1}\right|.$$

We can monitor the value of Δ to ensure that it stays below the error threshold.

If we are using an n^{th} order method, we know that $\Delta = ch^n$ for some c. The most efficient choice of step is that for which $\Delta = \epsilon$ (remember ϵ is our error threshold. Thus we would like to choose the new timestep, $\tilde{h}$ such that $c\tilde{h}^n = \epsilon$. But we know from the trial step that $\underset{\varsigma}{=}\Delta/h^n$. From this we can obtain the following rule to get the new timestep from the current step, the required error threshold and the local error estimated from the trial steps:

$$\tilde{h} = \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{8.16}$$

It is common to include a "safety factor", $\sigma_1 < 1$ to ensure that we stay a little below the error threshold:

$$\tilde{h}_1 = \sigma_1 \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{8.17}$$

Equally, since the solutions of ODE's are usually smooth, it is often sensible to ensure that the timestep does not increase or decrease by more than a factor of σ_2 (2 say) in a single step. To do this we impose the second constraint:

$$\begin{aligned} \tilde{h} &= \max\left\{\tilde{h}_1, \frac{h}{\sigma_2}\right\} \\ \tilde{h} &= \min\left\{\tilde{h}_1, \sigma_2 h\right\}. \end{aligned}$$

The improvement in performance obtained by using adaptive timestepping to integrate Eq. (8.15) with the simple forward Euler method is plotted as a function of the parameter μ in Fig. (8.2).

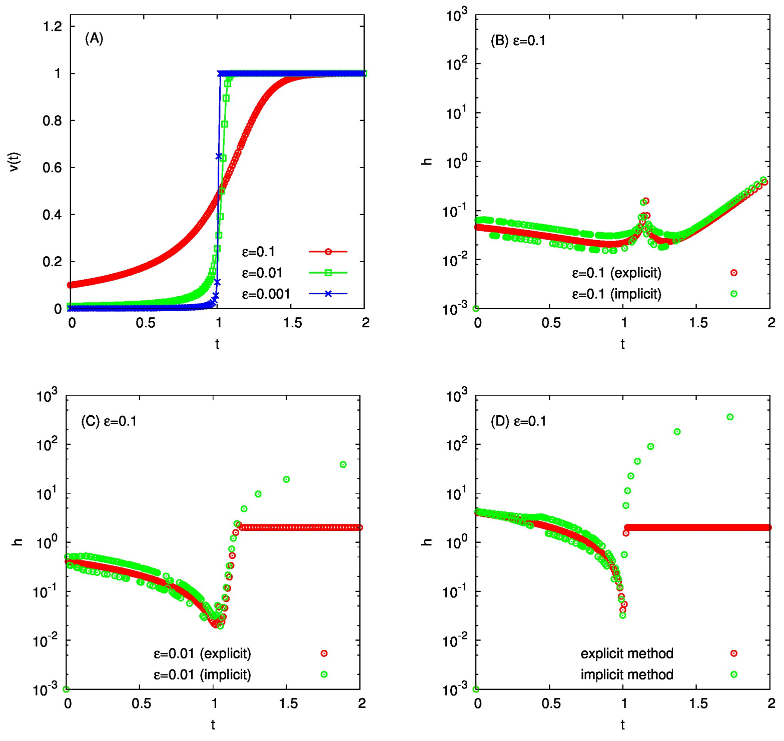

Figure 8.3: Comparison between behaviour of the forward and backward Euler schemes with adaptive timestepping applied to the problem in Eq. (8.19) for a range of values of ϵ.

Another situation in which adaptive timestepping is essential is when the ODE which we are trying to solve has a singularity. A singularity occurs at some time, $t = t^*$ if the solution of an ODE tends to infinity as $t \to t^*$. Naturally this will cause problems for any numerical integration routine. A properly functioning adaptive stepping strategy should reduce the timestep to zero as a singularity is approached. As a simple example, consider the equation,

$$\frac{du}{dt} = \lambda u^2.$$

with initial data $u(0) = u_0$. This equation is separable and has solution

$$u(t) = \frac{u_0}{1 - \lambda\, u_0\, t}.$$

This clearly has a singularity at $t^* = (\lambda\, u_0)^{-1}$.

8.5 Stiffness and implicit methods

Stiffness is an unpleasant property of certain problems which causes adaptive algorithms to select very small timesteps even though the solution is not changing very quickly. There is no uniformly accepted definition of stiffness. Whether a problem is stiff or not depends on the equation, the initial condition, the numerical method being used and the interval of integration. The common feature of stiff problems elucidated in a nice article by Moler [3], is that the required solution is slowly varying but “nearby” solutions vary rapidly. The archetypal stiff problem is the decay equation,

$$\frac{du}{dt} = -\lambda\, u \tag{8.18}$$

with $\lambda >> 1$. Efficient solution of stiff problems typically requires the use of implicit algorithms. Explicit algorithms with proper adaptive stepping will work but usually take an unfeasibly long time. Here is a more interesting nonlinear example taken from [3]:

$$\frac{dv}{dt} = v^2 - v^3. \tag{8.19}$$

with initial data $v(0) = \epsilon$. Find the solution on the time interval $[0, 2/\epsilon]$. Its solution is shown in Fig. 8.3(A). The problem becomes stiff as ϵ is decreased. We see, for a given error tolerance (in this case, a relative error threshold of 10^{-5}), that if $\epsilon \ll 1$ the implicit backward Euler scheme can compute the latter half of the solution (the stiff part) with enormously larger timesteps than the corresponding explicit scheme. An illuminating animation and further discussion of stiffness with references is available on Scholarpedia [4].

8.6 Runge-Kutta methods

We finish our discussion of methods for integration of ODEs with a brief discussion of the Runge-Kutta methods since these are the real workhorses of the business. When you call a black box integration routine in MatLab or Mathematica to integrate an ODE, odds are it is using a Runge-Kutta method so it is worth knowing a little about how they work. Runge-Kutta methods aim to retain the accuracy of the multistep predictor corrector methods but without having to use more than the current value of $\mathbf{u}$ to predict the next one - ie they are self-starting. The idea is roughly to make several predictions of the value of the solution at several points in the interval $[t_1, t_{i+1}]$ and then weight them cleverly so as to cancel errors.

An n^{th} order Runge–Kutta scheme for Eq. (8.4) looks as follows. We first calculate n estimated values of $\mathbf{F}$ which are somewhat like the predictors used in Sec. 8.3:

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{F}(\mathbf{u}_i)\\
\mathbf{f}_2 &= \mathbf{F}(\mathbf{u}_i + a_{2\,1}\,h\,\mathbf{f}_1)\\
\mathbf{f}_3 &= \mathbf{F}(\mathbf{u}_i + a_{3\,1}\,h\,\mathbf{f}_1 + a_{3\,2}\,h, \mathbf{f}_2)\\
&\vdots\\
\mathbf{f}_n &= \mathbf{F}(\mathbf{u}_i + a_{n\,1}\,h\,\mathbf{f}_1 + \ldots + a_{n\,n-1}\,h, \mathbf{f}_{n-1})
\end{aligned}$$

We then calculate $\mathbf{u}_{i+1}$ as

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,(b_1\mathbf{f}_1 + b_2\mathbf{f}_2 + \ldots + b_n\mathbf{f}_n). \tag{8.20}$$

The art lies in choosing the $a_{i\,j}$ and b_i such that Eq. (8.20) has a stepwise error of $O(h^{n+1})$. The way to do this is by comparing Taylor expansions as we did to determine the accuracy of the Improved Euler Method in Sec. 8.3 and choosing the values of the constants such that the requisite error terms vanish. It turns out that this choice is not unique so that there are actually parametric families of Runge-Kutta methods of a given order.

We shall derive a second order method explicitly for a scalar equation, EQ. (8.13). Derivations of higher order schemes provide nothing new conceptually but require a lot more algebra. Extension to the vector case is again straightforward but requires care with indices. Let us denote our general 2-stage Runge-Kutta algorithm by

$$\begin{aligned}
f_1 &= F(u_i)\\
f_2 &= F(u_i + a\,h\,f_1)\\
u_{i+1} &= u_i + h\,(b_1 f_1 + b_2 f_2).
\end{aligned}$$

Taylor expand f_2 up to second order in h:

$$\begin{aligned}
f_2 &= F(u_i) + a\,h\,F_i\,\frac{dF}{du}(u_i) + O(h^2)\\
&= F_i + a\,h\,F_i\,F_i' + O(h^2)
\end{aligned}$$

Our approximate value of u_{i+1} is then

$$\widetilde{u}_{i+1} = u_i + (b_1 + b_2)hF_i + ab_2h^2F_i\,F_i' + O(h^3).$$

If we compare this to the usual Taylor expansion of u_{i+1} we see that we can make the two expansions identical up to $O(h^3)$ if we choose

$$\begin{aligned}
b_1 + b_1 &= 1\\
ab_2 &= \frac{1}{2}.
\end{aligned}$$

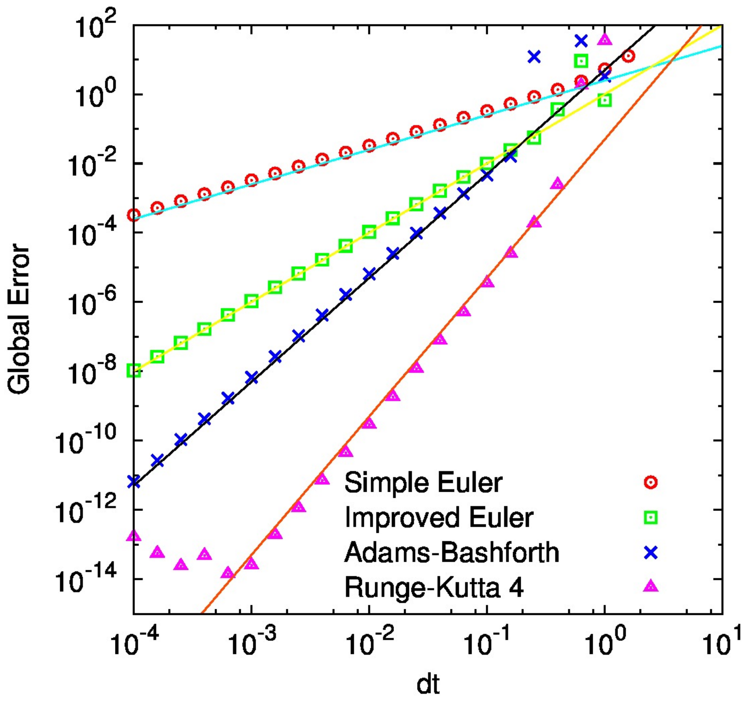

Figure 8.4: Log-Log plot showing the behaviour of the global error as a function of step size for several different timestepping algorithms applied to Eq. (8.23). The straight lines show the theoretically expected error, h^n, where $n = 1$ for the simple Euler method, $n = 2$ for the improved Euler method, $n = 3$ for the Adams–Bashforth method and $n = 4$ for the 4th order Runge–Kutta method.

A popular choice is $b_1 = b_2 = \frac{1}{2}$ and $a = 1$. This gives, what is often considered the "standard" second order Runge-Kutta scheme:

$$\begin{aligned} f_1 &= F(u_i) \\ f_2 &= F(u_i + h\, g_1) \\ u_{i+1} &= u_i + \frac{h}{2}\,(f_1 + f_2). \end{aligned} \tag{8.21}$$

Perhaps it looks familiar. The standard fourth order Runge–Kutta scheme, with a stepwise error of $O(h^5)$, is really the workhorse of numerical integration since it has a very favourable balance of accuracy, stability and efficiency properties. It is often the standard choice. We shall not derive it but you would be well advised to use it in your day-to-day life. It takes the following form:

$$\begin{aligned} \mathbf{f}_1 &= \mathbf{F}(\mathbf{u}_i) \\ \mathbf{f}_2 &= \mathbf{F}(\mathbf{u}_i + \frac{h}{2}\,\mathbf{f}_1) \\ \mathbf{f}_3 &= \mathbf{F}(\mathbf{u}_i + \frac{h}{2}\,\mathbf{f}_2) \\ \mathbf{f}_4 &= \mathbf{F}(\mathbf{u}_i + h\,\mathbf{f}_3) \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \frac{h}{6}(\mathbf{f}_1 + 2\mathbf{f}_2 + 2\mathbf{f}_3 + \mathbf{f}_4). \end{aligned} \tag{8.22}$$

Fig. 8.4 compares the error in several of the integration routines discussed above when applied to the test problem

$$\frac{d^2v}{dt^2} + 2\,t\frac{dv}{dt} - v = 0, \tag{8.23}$$

with initial conditions $v(0) = 0$, $\frac{dv}{dt}(0) = \frac{2}{\sqrt{\pi}}$. With these initial conditions, Eq. (8.23) has a simple exact solution:

$$v(t) = \mathrm{Erf}(t), \tag{8.24}$$

where $\mathrm{Erf}(x)$ is defined as

$$\mathrm{Erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-y^2}\, dy. \tag{8.25}$$

Notice that the superior accuracy of the RK4 method very quickly becomes limited by round-off error. The smallest stepsize below which the error starts to get worse as the stepsize is decreased is quite modest - about 10^{-3} in this example.

8.7 Continuous Fourier Transform

The Fourier transform is used to represent a function as a sum of constituent harmonics. It is a linear invertible transformation between the time-domain representation of a function, which we shall denote by $h(t)$, and the frequency domain representation which we shall denote by $H(f)$. In one dimension, the Fourier transform pair consisting of the forward and inverse transforms, is often written as

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{i\,f\,t}\, dt \\ h(t) &= \frac{1}{2\,\pi} \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-i\,f\,t}\, df. \end{aligned}$$

Note that the Fourier transform is naturally defined in terms of complex functions. Both $h(t)$ and $H(f)$ are, in general, complex-valued. Normalisation can be a thorny issue however and there are many different conventions in different parts of the literature. In fact the transform pair

$$\begin{aligned} H(f) &= \sqrt{\frac{|b|}{(2\,\pi)^{1-a}}} \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{i\,b\,f\,t}\, dt \\ h(t) &= \sqrt{\frac{|b|}{(2\,\pi)^{1+a}}} \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-i\,b\,f\,t}\, df. \end{aligned}$$

is equally good for any real values of the parameters a and b. In this module we shall adopt the convention $a = 0$ and $b = 2\,\pi$, which is common in the signal processing literature. With this choice the Fourier transform pair is:

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{2\,\pi\,i\,f\,t}\, dt \\ h(t) &= \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-2\,\pi\,i\,f\,t}\, df. \end{aligned} \tag{8.26}$$

The Fourier transform has several important symmetry properties which are often useful. The following properties are easily shown by direct calculation based on the definition:

- Parity is preserved: if $h(t)$ is even/odd then $H(f)$ is even/odd.
- Fourier transform of a real function: if $h(t)$ is real valued then $H(f)$, while still complex-valued, has the following symmetry:

$$H(-f) = H(f)^*, \tag{8.27}$$

  where $H(f)^*$ denotes the complex conjugate of $H(f)$.

Sometimes it is possible to compute the integral involved in Eq. (8.26) analytically. Consider the important example of a Gaussian function:

$$h(t) = \frac{1}{\sqrt{2\,\pi\,\sigma^2}}\, \mathrm{e}^{-\frac{t^2}{2\,\sigma^2}}.$$

The Fourier transform can be calculated analytically using a standard trick which involves completing the square in the exponent of a Gaussian integral. From Eq. (8.26) we have

$$\begin{aligned}
H(f) &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{t^2}{2\,\sigma^2}}\,\mathrm{e}^{2\,\pi\,i\,f\,t}\,dt \\
&= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}\left[t^2-4\pi i\sigma^2 t f\right]}dt \\
&= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}\left[t^2-2(2\pi i\sigma^2 f)t+(2\pi i\sigma^2 f)^2-(2\pi i\sigma^2 f)^2\right]}dt \\
&= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \left[\int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}(t-2\pi i\sigma^2 f)^2}dt\right]\,\mathrm{e}^{\frac{1}{2\,\sigma^2}\,(2\pi i\sigma^2 f)^2} \\
&= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \left[\sqrt{2\,\pi\,\sigma^2}\right]\,\mathrm{e}^{-2\pi^2\sigma^2 f^2} \\
&= \mathrm{e}^{-2\pi^2\sigma^2 f^2}.
\end{aligned}$$

Under the Fourier transform, the Gaussian function is mapped to another Gaussian function with a different width. If σ^2 is large/small then $h(t)$ is narrow/broad in the time domain. Notice how the width is inverted in the frequency domain. This is an example of a general principle: signals which are strongly localised in the time domain are strongly delocalised in the frequency domain and vice versa. This property of the Fourier transform is the underlying reason for the "Uncertainty Principle" in quantum mechanics where the "time" domain describes the position of a quantum particle and the "frequency" domain describes its velocity.

It is somewhat exceptional that the Fourier transform turns out to be a real quantity. In general, the Fourier transform, $H(f)$, of a real function, $h(t)$, is still complex. In fact, the Fourier transform of the Gaussian function is only real-valued because of the choice of the origin for the t-domain signal. If we would shift $h(t)$ in time, then the Fourier tranform would have come out complex. This can be seen from the following translation property of the Fourier transform. Suppose we define $g(t)$ to be a shifted copy of $h(t)$:

$$g(t) = h(t+\tau).$$

Writing both sides of this equation in terms of the Fourier transforms we have

$$\int_{-\infty}^{\infty} G(f)\,\mathrm{e}^{-2\,\pi\,i\,f\,t}\,df = \int_{-\infty}^{\infty} H(f)\,\mathrm{e}^{-2\,\pi\,i\,f\,(t+\tau)}\,df,$$

from which we conclude that

$$G(f) = \mathrm{e}^{-2\pi i \tau}\,h(f). \tag{8.28}$$

Translation in the time-domain therefore corresponds to multiplication (by a complex number) in the frequency domain.

8.7.1 Power Spectrum

Since Fourier transforms are generally complex valued, it is convenient to summarise the spectral content of a signal by plotting the squared absolute value of $H(f)$ as a function of f. This quantity,

$$P(f) = H(f)\,H^*(f), \tag{8.29}$$

is called the power spectrum of $h(t)$. The power spectrum is fundamental to most applications of Fourier methods.

8.7.2 Convolution Theorem

The convolution of two functions $f(t)$ and $g(t)$ is the function $(f*g)(t)$ defined by

$$(f*g)(t) = \int_{-\infty}^{\infty} f(\tau)\,g(t+\tau)\,d\tau. \tag{8.30}$$

Convolutions are used very extensively in time series analysis and image processing, for example as a way of smoothing a signal or image. The Fourier transform of a convolution takes a particularly simple form. Expressing f and g in terms of their Fourier transforms in Eq. (8.30) above, we can write

$$\begin{aligned}(f*g)(t) &= \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty}F(f_1)\,\mathrm{e}^{-2\,\pi\,i\,f_1\,\tau}\,df_1\right]\left[\int_{-\infty}^{\infty}g(f_2)\,\mathrm{e}^{-2\,\pi\,i\,f_2\,(t+\tau)}\,df_2\right]\,d\tau\\ &= \int_{-\infty}^{\infty}df_1\int_{-\infty}^{\infty}df_2\,F(f_1)\,G(f_2)\,\mathrm{e}^{-2\pi i f_1}\int_{-\infty}^{\infty}d\tau\mathrm{e}^{-2\pi i(f_1+f_2)}\\ &= \int_{-\infty}^{\infty}df_1\int_{-\infty}^{\infty}df_2\,F(f_1)\,G(f_2)\,\mathrm{e}^{-2\pi i f_1}\,\delta(f_1+f_2)\\ &= \int_{-\infty}^{\infty}df_1 F(f_1)\,G^*(f_1)\,\mathrm{e}^{-2\pi i f_1}.\end{aligned}$$

Thus the Fourier transform of the convolution of f and g is simply the product of the individual Fourier transforms of f and g. This fact is sometimes called the Convolution Theorem. One of the most important uses of convolutions is in time-series analysis. The convolution of a signal, $h(t)$, (now thought of as a time-series) with itself is known as the auto-correlation function (ACF) of the signal. It is usually written

$$R(\tau) = \int_{-\infty}^{\infty} h(t+\tau)\,h(t)\,dt. \tag{8.31}$$

The ACF quantifies the memory of a time-series. Using the Convolution Theorem we can write

$$R(\tau) = \int_{-\infty}^{\infty} df H(f)\,H^*(f)\,\mathrm{e}^{-2\pi i f} = \int_{-\infty}^{\infty} df P(f)\,\mathrm{e}^{-2\pi i f}. \tag{8.32}$$

The ACF of a signal is the inverse Fourier transform of the power spectrum. This fundamental result is known as the Wiener-Kinchin Theorem. It reflects the fact that the frequency domain and time-domain representations of a signal are just different ways of looking at the same information.

The Convolution Theorem means that convolutions are very easy to calculate in the frequency domain. In order for this to be practically useful we must be able to do the integral in Eq. (8.26) required to return from the frequency domain to the time domain. As usual, the set of problems for which this integral can be done using analytic methods is a set of measure zero in the space of all potential problems. Therefore a computational approach to computing Fourier transforms is essential. Such an approach is provided by the Discrete Fourier Transform which we now discuss.

8.8 Discrete Fourier Transform

8.8.1 Time-domain discretisation and the Sampling Theorem

In applications, a time-domain signal, $h(t)$, is approximated by a set of discrete values, $\{h_n = h(t_n),\ n = \ldots -2, -$ of the signal measured at a list of consecutive points $\{t_n,\ n = \ldots -2, -1, 0, 1, 2, \ldots\}$. We shall assume that the sample points are uniformly spaced in time: $t_n = n\,\Delta$. The quantity Δ is called the sampling interval and its reciprocal, $1/\Delta$, is the sampling rate. The quantity

$$f_c = \frac{1}{2\Delta}. \tag{8.33}$$

is known as the Nyquist frequency. The importance of the Nyquist frequency will become apparent below.

Clearly, the higher the sampling rate, the more closely the sampled points, $\{h_n\}$, can represent the true underlying function. Of principal importance in signal processing is the question of how high the sampling rate should be in order to represent a given signal to a given degree of accuracy. This question is partially answered by a fundamental result known as the Sampling Theorem (stated here without derivation):

Theorem 8.8.1. *Let $h(t)$ be a function and $\{h_n\}$ be the samples of $h(t)$ obtained with sampling interval Δ. If $H(f) = 0$ for all $|f| \geq f_c$ then $h(t)$ is* completely *determined by its samples. An explicit reconstruction is provided by the Whittaker-Shannon interpolation formula:*

$$h(t) = \Delta \sum_{n=-\infty}^{\infty} h_n \frac{\sin\left[2\pi f_c \left(t - n\Delta\right)\right]}{\pi \left(t - n\Delta\right)}.$$

This theorem explains why the Nyquist frequency is important. There are two aspects to this. On the one hand, if the signal $h(t)$ is such that the power spectrum is identically zero for large enough absolute frequencies then the Sampling Theorem tells us that it is possible to reconstruct the signal perfectly from its samples. The Nyquist frequency tells us the minimal sampling rate which is required in order to do this. On the other hand, we know that most generic signals have a power spectrum whose tail extends to all frequencies. For generic signals, the amount of power outside the Nyquist interval, $[-f_c, f_c]$, limits the accuracy with which we can reconstruct a signal from its samples. The Nyquist frequency is then used in conjunction with an error tolerance criterion to decide how finely a function must be sampled in order to reconstruct the function from its samples to a given degree of accuracy.

8.8.2 Aliasing

The above discussion about using the Nyquist frequency to determine the sampling rate required to reconstruct a signal to a given degree of accuracy implicitly assumes that we know the power spectrum of $h(t)$ to begin with in order to determine how much power lies outside the Nyquist interval. In practice, we usually only have access to the samples of $h(t)$. From the samples alone, it is in principle impossible to determine how much power is outside the Nyquist interval. The reason for this is that the process of sampling moves frequencies which lie outside of the interval $[-f_c, f_c]$ into this interval via a process known as aliasing.

Aliasing occurs because two harmonics $\mathrm{e}^{2\pi i f_1 t}$ and $\mathrm{e}^{2\pi i f_2 t}$ give the same samples at the points $t_n = n\Delta$ if $f_1 - f_2$ is an integer multiple of Δ^{-1} (which is the width of the Nyquist interval). This can be seen by direct computation:

$$\begin{aligned}
&\mathrm{e}^{2\pi i f_1 n\Delta} = \mathrm{e}^{2\pi i f_1 n\Delta} && \forall n\\
\Rightarrow &\mathrm{e}^{2\pi i (f_1 - f_2) n\Delta} = 1 && \forall n\\
\Rightarrow &\mathrm{e}^{2\pi i (f_1 - f_2)\Delta} = 1 &&\\
\Rightarrow &(f_1 - f_2)\,\Delta = m && m \in \mathbb{Z}\\
\Rightarrow &(f_1 - f_2) = \frac{m}{\Delta} && m \in \mathbb{Z}
\end{aligned}$$

Due to aliasing, it is impossible to tell the difference between the signals $h_1(t) = \mathrm{e}^{2\pi i f_1 t} + \mathrm{e}^{2\pi i f_2 t}$ and $h_2(t) = 2\,\mathrm{e}^{2\pi i f_1 t}$ from the samples alone. Aliasing means that under-sampled signals cannot be identified as such from looking at the sampled values. Care must therefore be taken in practice ensure that the sampling rate is sufficient to avoid aliasing errors.

8.8.3 Derivation of the Discrete Fourier Transform

Let us suppose we now have a finite number, N, of samples of the function $h(t)$: $\{h_n = h(t_n),\ n = 0, \ldots N-1\}$. For convenience we will assume that N is even. We now seek to use these values to estimate the integral in Eq. (8.26) at a set of discrete frequency points, f_n. What values should we choose for the f_n? We know from the Sampling Theorem that if the sampling rate is sufficiently high, the Nyquist interval contains the information necessary to reconstruct $h(t)$. If the sampling rate is too low, the power outside the Nyquist interval will be mapped back into the Nyquist interval by aliasing. In either case, we should choose our frequency points to discretise the Nyquist interval. Let us do this uniformly by choosing $f_n = \frac{n}{\Delta N}$ for $n = -\frac{N}{2}, \ldots + \frac{N}{2}$. The extremal frequencies are clearly $\pm f_c$. Those paying close attention will notice that there are $N+1$ frequency points to be estimated from N samples. It will turn out that not all $N+1$ frequencies are independent. We will return to this point in a moment.

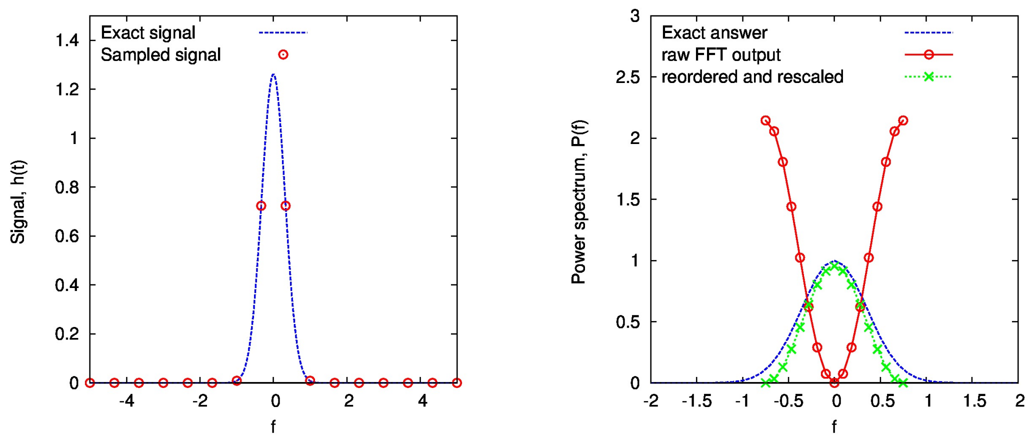

Figure 8.5: Discrete Fourier Transform of a Gaussian sampled with 16 points in the domain [-5:5] using the fftw3 implementation of the Fast Fourier Transform.

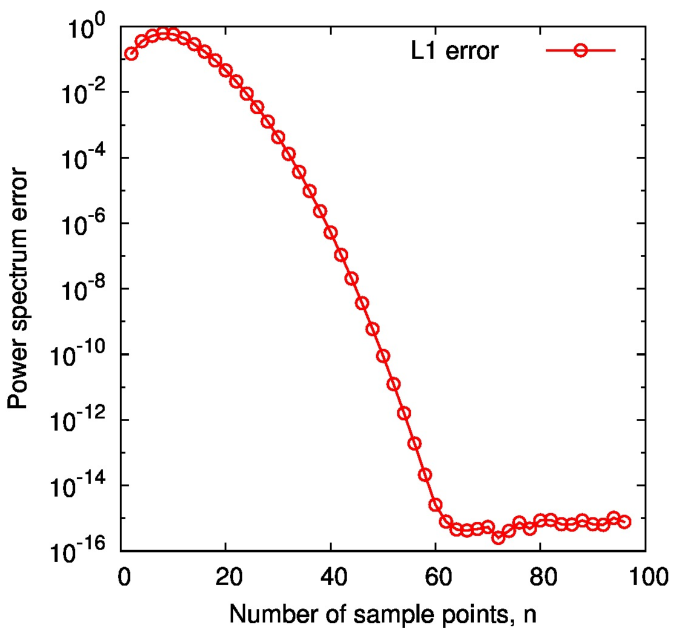

Figure 8.6: L1 error in the power spectrum of a Gaussian approximated using the Discrete Fourier Transform plotted as a function of the number of sample points in the domain [-5:5]. The fftw3 implementation of the Fast Fourier Transform was used.

Let us now estimate the integral $H(f)$ in Eq. (8.26) at the discrete frequency points f_n by approximating the integral by a Riemann sum:

$$\begin{aligned} H(f_n) &\approx \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{2\pi i f_n t_k} \Delta \\ &= \Delta \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{2\pi i \left(\frac{n}{N\Delta}\right)(k\Delta)} \\ &= \Delta \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \\ &\equiv \Delta \, H_n, \end{aligned}$$

where the array of numbers, $\{H_n\}$, defined as

$$H_n = \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \tag{8.34}$$

is called the Discrete Fourier Transform (DFT) of the array $\{h_n\}$. The inverse of the transform is obtained by simply changing the sign of the exponent. Note that the DFT is simply a map from one array of complex numbers to another. All information about the time and frequency scales (including the sampling rate) have been lost. Thought of as a function of the index, n, the DFT is periodic with period N. That is

$$H_{n+N} = H_n.$$

We can see this by direct computation:

$$\begin{aligned} H_{n+N} &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i (n+N) k}{N}} \\ &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \, \mathrm{e}^{2\pi i k} \\ &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \\ &= H_n. \end{aligned}$$

We can therefore shift the index, n, used to define the frequencies f_n, by $\frac{N}{2}$ in order to obtain an index which runs over the range $n = 0 \ldots N$ rather than $n = -\frac{N}{2}, \ldots + \frac{N}{2}$. This is more convenient for a computer where arrays are usually indexed by positive numbers. Most implementations of the DFT adopt this convention. One consequence of this convention is that if we want to plot the DFT as a function of the frequencies (for example to compare against an analytic expression for the corresponding continuous Fourier transform) we need to obtain the negative frequencies from $H_{-n} = HN - n$. Another consequence of this periodicity is that $H_{-\frac{N}{2}} = H_{\frac{N}{2}}$ so $H_{\frac{N}{2}}$ corresponds to both frequencies $f = \pm f_c$ at either end of the Nyquist interval. Thus, as mentioned above, only N of the $N+1$ values of n chosen to discretise the Nyquist interval are independent. THe DFT, Eq. (8.34), is therefore a map between arrays of length N which can be written as a matrix multiplication:

$$H_n = \sum_{k=0}^{N-1} W_{nk} \, h_k, \tag{8.35}$$

where W_{pq} are the elements of the $N \times N$ matrix

$$W_{pq} = \mathrm{e}^{\frac{2\pi i p q}{N}}.$$

The application of the DFT to a sampled version of the Gaussian function is shown in Fig. 8.5. The frequency domain plot on the right of Fig. 8.5 shows the power spectrum obtained from the output of the DFT plotted simply as a function of the array index (in red). Notice how the frequencies appear in the "wrong" order as a result of the shift described above. Notice also how the amplitude of the power spectrum obtained from the DFT is significantly larger than the analytic expression obtained for the continuous Fourier transform. This is due to the fact that the pre-factor of Δ in the approximation of $H(f)$ derived above is not included in the definition of the DFT. When both of these facts are taken into account by rescaling and re-ordering the output of the DFT, the green points are obtained which are a much closer approximation to the true power spectrum. The difference between the two is due to the low sampling rate. As the sampling rate is increased the approximation obtained from the DFT becomes better. This is illustrated in Fig. 8.6 which plots the error as a function of the number of sample points. Why do you think the error saturates at some point?

8.9 The Fast Fourier Transform algorithm

From Eq. (8.35), one might naively expect that the computation of the DFT is an $O(N^2)$ operation. In fact, due to structural regularities in the matrix W_{pq} it is possible to perform the computation in $O(N \log_2 N)$ operations using a divide-and-conquer algorithms which is known as the Fast Fourier Transform (FFT). In all that follows we will assume that the input array has length $N = 2^L$ elements. The FFT algorithm is based on the Danielson-Lanczos lemma which states that a DFT of length N can be written as a sum of 2 DFTs of length $N/2$. This can be seen by direct computation where the trick is to build the sub-transforms of lenght $N/$ from the elements of the original transform whose indices are odd/even respectively:

$$\begin{aligned}
H_k &= \sum_{j=0}^{N-1} h_j \, \mathrm{e}^{\frac{2\pi i n j}{N}} \\
&= \left[\sum_{j=0}^{\frac{N}{2}-1} h_{2j} \, \mathrm{e}^{\frac{2\pi i n (2j)}{N}}\right] + \left[\sum_{j=0}^{\frac{N}{2}-1} h_{2j+1} \, \mathrm{e}^{\frac{2\pi i n (2j+1)}{N}}\right] \\
&= \left[\sum_{j=0}^{\frac{N}{2}-1} h_j^{(e)} \, \mathrm{e}^{\frac{2\pi i n j}{N/2}}\right] + \mathrm{e}^{\frac{2\pi i k}{N}} \left[\sum_{j=0}^{\frac{N}{2}-1} h_j^{(o)} \, \mathrm{e}^{\frac{2\pi i n j}{N/2}}\right].
\end{aligned}$$

where $\left\{h_j^{(o/e)}, j = 0, \ldots \frac{N}{2} - 1\right\}$ are the arrays of length $\frac{N}{2}$ constructed from the odd/even elements of the original input array respectively. The last line is simply a linear combination of the DFTs of these two arrays:

$$H_k = H_k^{(e)} + \alpha_k \, H_k^{(o)}, \tag{8.36}$$

where $\alpha_k = \mathrm{e}^{\frac{2\pi i k}{N}}$. Note that since the index k on the LHS ranges over $k = 0, \ldots, N-1$, whereas the arrays on the RHS are only of length $N/2$, the periodicity of the DFT must be invoked to make sense of this formula. The FFT algorithm works by recognising that this can be applied recursively. After $L = \log_2 N$ subdivisions, we are left with arrays of length 1, each of which contains one of the elements of the original input array. Thus at the lowest level of the recursion, we just have a re-ordered copy of the original array. This is illustrated schematically in Fig. 8.7. The DFT of an array of length one is trivially just the array itself. The DFT of the original array can therefore be reassembled by going back up the tree linearly combining pairs of arrays of length n at each level to form arrays of length $2n$ at the next level up according to the rule given in Eq. (8.36 until we arrive back at the top level. At each level, there are clearly N operations required and there are $L = \log_2 N$ levels. Therefore the total complexity of this process is $O(N \, \log_2 N)$.

The key insight of the FFT algorithm is to construct the re-ordered array first and then go back up the tree. Notice how each position in the array at the bottom level can be indexed by a string of e's and o's depending on the sequence of right and left branches which are followed when descending the tree to reach that position. This sequence can be reversed to find out which element of the input array should be placed at which position at the bottom of the recursion tree. This is done using the

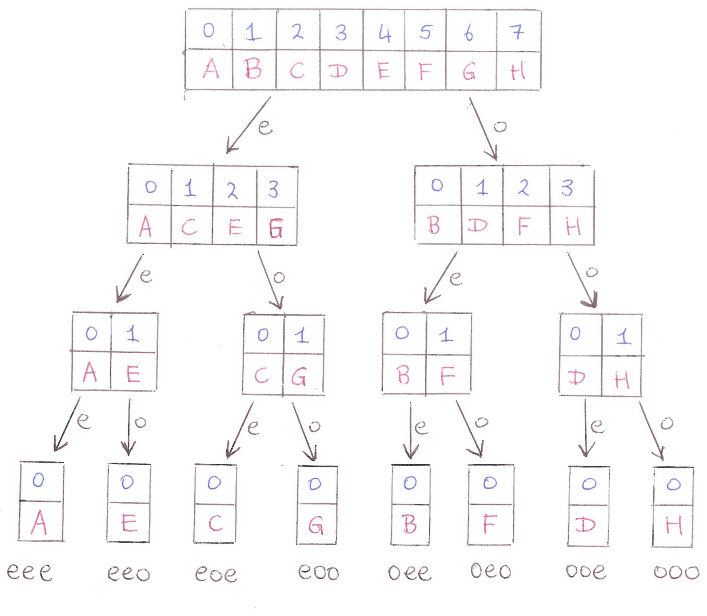

Figure 8.7: Graphical illustration of the recursive structure of the FFT algorithm for an array of length 8. At each level, each array is divided into two sub-arrays containing the elements with odd and even indices respectively. At the lowest level, the position of an element is indexed by a unique string of e's and o's. This string is determined by the sequence of right and left branches which are followed when descending the tree to reach that position. The bit reversal trick then identifies which element of the original array ends up in each position.

bit-reversal rule. Starting with the string corresponding to a particular position at the bottom of the recursion tree we proceed as follows:

1. reverse the order of the string of e's and o's.
2. replace each e with 0 and each o with 1.
3. interpret the resulting string of zeros and ones as the binary representation of an integer.

The integer obtained in this way is the index of the element in the original input array which should be placed at this position at the bottom of the recursion tree. It is worth convincing oneself that this really works by checking some explicit examples in Fig. 8.7. This re-ordering can be done in $O(N)$ operations.

Bibliography

[1] Linear multistep method, 2014. `http://en.wikipedia.org/wiki/Linear_multistep_method`.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[3] C. Moler. Matlab news & notes may 2003 - stiff differential equations, 2003. `http://www.mathworks.com/company/newsletters/news\_notes/clevescorner/may03\_cleve.html`.

[4] L. F. Shampine and S. Thompson. Stiff systems. Scholarpedia, 2(3):2855, 2007.