MA50174 ADVANCED NUMERICAL METHODS

C.J.Budd

Contents

Chapter 6

Initial Value Problems (IVPs)

6.1 Introduction

An initial value problem is an ordinary differential equation of the form

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u}) \quad \text{where} \quad \mathbf{u}(t_o) = \mathbf{u}_o \quad \text{is given and} \quad \mathbf{u} \in I\!R^n. \tag{6.1}$$

In the special case when $\mathbf{f}(t, \mathbf{u}) = \mathbf{f}(t)$ we have

$$\mathbf{u}(t) = \int_{to}^{t} \mathbf{f}\, dt + \mathbf{u}_o,$$

and the task of evaluating this integral accurately is called *quadrature* To solve any differential equation we need to put it into the standard form given by (6.1). Any equation involving higher derivatives can be reformulated as such a vector equation. For example, if w satisfies the second order differential equation,

$$\frac{d^2w}{dt^2} = w$$

Let

$$u_1 = w, \quad u_2 = dw/dt.$$

Then

$$\begin{aligned} du_1/dt &= u_2, \\ du_2/dt &= u_1. \end{aligned}$$

Thus if

$$\mathbf{u} = (u_1, u_2)^T$$

we have

$$\frac{d\mathbf{u}}{dt} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \mathbf{u} \equiv A\mathbf{u}$$

Initial value problems (IVPs) come in various forms and there is no such thing as a perfect all purpose IVP solver. MATLAB offers you quite a choice. Will try to show you how to choose which one to use for a given problem.

We say that an ODE problem is

1. *Linear* if $\mathbf{f}(t, \mathbf{u})$ is linear in $\mathbf{u}$
2. *Autonomous* if $\mathbf{f}(t, \mathbf{u}) \equiv \mathbf{f}(\mathbf{u})$
3. *Non-stiff* if all components of the equation evolve on the same timescale. This occurs (roughly) if the Jacobian matrix $\partial\mathbf{f}/\partial\mathbf{u}$ has all its eigenvalues of similar size.

4. *Stiff* if different components of the system evolve on different time scales. These are very common in chemical reactions with reactions going on at different rates and in ODEs resulting from spatial discretisations of PDEs. They also occur in PDEs where different modes (Fourier modes) evolve at very different rates. Stiff problems are much harder to solve numerically than non-stiff ones.

5. *Hamiltonian* if $\mathbf{f}$ takes the form
$$\mathbf{f} = J^{-1}\nabla H$$
where $H(\mathbf{y})$ is the Hamiltonian of the system and
$$J = \begin{bmatrix} 0 & -I \\ I & 0 \end{bmatrix} \quad \text{where } I \text{ is the } \tfrac{n}{2} \times \tfrac{n}{2} \text{ identity matrix.}$$
Hamiltonian equations arise very commonly in rigid body mechanics, celestial mechanics (astronomy) and molecular dynamics. To solve a Hamiltonian equation accurately over long time periods we must use special numerical methods, such as symplectic or reversible methods.

All modern software for IVPs is a combination of three components

- The actual solver.
- A way of estimating the error of the solution.
- A step-size control mechanism.

Thus, the IVP solver attempts to use the best method to keep the (estimated) error within a prescribed tolerance. Whilst it is *essential* to have some form of error control, no such method is infallible. The numerical solution of IVPs is well covered in many texts, for example A.Iserles "A first course in the numerical solution of differential equations."

6.2 Quadrature

Suppose that $f(t)$ is an arbitrary function, how accurately can we find
$$u = \int_a^b f(t)dt \; ? \tag{6.2}$$
MATLAB determines this integral approximately by using a composite Simpson's rule.The idea behind this is as follows:
Suppose we take an interval of length $2h$, without loss of generality this is the interval $[0, 2h]$.

- *Evaluate* f at the points $0, h, 2h$
- Then *approximate* f by a parabola through these points, by using a quadratic interpolant as in Chapter 3.
- *Integrate* this approximation to get an estimate for the integral of f.

This gives
$$\int_0^{2h} f dx \approx \frac{h}{6}\left[f(0) + 4f(h) + f(2h)\right] \equiv S_h$$
This approximation is unreasonably accurate. It can be shown (see Froberg, "Numerical Analysis") that
$$|S_h - \int_0^{2h} f(t)dt| = \frac{h^5}{90}\,|f^{(iv)}(\xi)| \quad \text{where} \quad 0 < \xi < 2h \tag{6.3}$$

The *local* error is proportional to h^5 and to $f^{(iv)}$. The *traditional* use of Simpson's rule to evaluate (6.2) over $[a, b]$ breaks this interval into sub-intervals of length $2h$ takes h constant between a and b and adds up the results to get a total error estimated by

$$\frac{h^4}{90}\,|b-a|\max(|f^{(iv)}|).$$

MATLAB is more intelligent than this and it uses an *adaptive* version of Simpson's rule. In this procedure h is chosen carefully over each interval to keep the error estimated by (6.2) less than a user specified tolerance. In particular h is small when f is varying more rapidly and $f^{(iv)}$ is large. The integrals over each sub-interval are then combined to give the total. This method is especially effective if f has a singularity.
The procedure uses the instruction

$$> i = quad(\text{@fun}, a, b, \text{tol})$$

where fun is the function to be integrated and tol is the tolerance. There is another MATLAB code > *quadl*. This approximates f by a higher order polynomial. This is more accurate if f is smooth but unreliable if f has singularities.

6.3 Non-stiff ordinary differential equations (ODEs)

A non-stiff ordinary differential equation has components which *all* evolve on similar time-scales. They are the easiest differential equations to solve by using a numerical method. In particular they can often be solved by using explicit methods that do not require the solution of nonlinear equations.

6.3.1 The Forward Euler Method

The oldest, easiest to apply and analyse, method for such problems is the explicit forward *Euler* method. Suppose that $\mathbf{u}$ satisfies the ODE

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u}(0) = \mathbf{u}_0$$

- We take a small step h and approximate $\mathbf{u}((n-1)h)$ by $\mathbf{U}^n$
- We set $\mathbf{U}^1 = \mathbf{u}_0$
- Now for each successive n we update $\mathbf{U}_n$ through

$$\mathbf{U}^{n+1} = \mathbf{U}^n + h\mathbf{f}(t, \mathbf{U}^n) = \mathbf{U}^n + h\mathbf{f}((n-1)h, \mathbf{U}^n) \qquad (6.4)$$

This method is *easy to use* and each step is *fast* as no equations need to be evaluated and there is only *one* function evaluation per step. The Forward Euler method is still used when $\mathbf{f}$ is hard to evaluate and there are a large number of simultaneous equations. Problems of this kind arise in weather forecasting. The main problem with this method is that there are often severe restrictions on the size of h.

Local Error.
At each stage of the method *a small error* is made. These errors accumulate over successive intervals to give an overall error. If the exact solution $\mathbf{u}(t)$ is substituted into the equation (6.4), there is a mismatch E between the two sides, called the *local truncation error* or LTE. This is a good estimate for the error made by the method at each step. It is estimated by

$$|E| = |\mathbf{u}(nh) - \mathbf{u}((n-1)h) - hf((n-1)h, \mathbf{u}((n-1)h))| = \frac{h^2}{2}\,|\mathbf{u}''(\xi)| \qquad \text{where} \quad (n-1)h < \xi < nh \quad (6.5)$$

As in quadrature the local truncation error is proportional to a power of h, in this case h^2 and a higher derivative of u, in this case u''. So, if u'' is large (rapid change), we must take h small to give a small error. The smaller the value of h is, the more accurate the answer will be. (We will see later that h is also restricted by stability considerations).

Global Error
Each time the method is applied an error is made and the errors accumulate over all the calculations. If we want to approximate $\mathbf{u}(T)$ then need to make $T/h+1 \equiv N$ calculations so that $\mathbf{U}^N \approx \mathbf{u}(T)$. We will call ε the global error if

$$\varepsilon = |\mathbf{U}^N - \mathbf{u}(T)|, \quad N = \frac{T}{h} + 1.$$

This can be crudely estimated by

$$\varepsilon \approx \frac{Nh^2}{2}\, max|\mathbf{u}''| \approx \mathbf{Th}\, max|\mathbf{u}''|.$$

We note that the overall error is proportional to h and to $max|\mathbf{u}''|$. We say that this is an $O(h)$ or a *first order* method.

The estimate we have obtained implies that the global error grows in proportion to both h and T. In fact, this is rarely observed. In the worst case for some problems the error grows in proportion to e^T, when looking at periodic problems it can grow like T^2 and for problems with a lot of symmetry, the errors can accumulate much more slowly so that the overall error may not grow at all with T.

We start by looking at the worst case analysis.
We say that f is globally Lipshitz if there is a constant L such that

$$|f(t,y) - f(t,z)| < L|y-z| \tag{6.6}$$

It can then be shown (see A. Iserles "A first course in the numerical solution of differential equations") that if $|u''| \le M$ then the global error has an upper bound given by

$$\varepsilon \le \frac{hM}{2L}\left(e^{LT} - 1\right) \tag{6.7}$$

If T is *fixed* and $h \to 0$ then the upper bound given in (6.7) shows that ε is proportional to h in this limit. In practice the Forward Euler method is rarely used in this way. We often take h fixed and let $T \to \infty$. This is very dangerous as errors can accumulate rapidly making the method unusable. In this case we need a more careful control on the global growth of errors. This is the philosophy behind geometric integration based methods.

Variable time-stepping
As in the quadrature process, Euler's method can be used with a *variable* step size. This allows some control over the local error. To do this we take a sequence of (small) time steps, so that the solution is approximated at times t_k with so that $U^k \approx u(t_k)$

$$h_k \equiv t_{k+1} - t_k.$$

If we have an estimate for $u''(\xi),\ \ \xi \in [t_k, t_{k+1}]$ then h_k is chosen so that at each step

$$E = \frac{h_k^2 |u''|}{2} < \text{Tol}$$

where TOL is specified by the user. Of course, without knowing $u(t)$ we do not know u'' in general. We can either estimate an upper bound a -priori to the calculation or we can try to deduce its value (a-posteriori) from the numerical calculation. This latter procedure often uses an error estimate call the *Milne device.* (See Iserles.)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%   Example 1: Use of the Forward Euler Method                %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  Consider the ODE  du/dt = -2t u^2,  u(0) = 1               %
%                    ==========================               %
%                                                             %
%  This has the solution  u(t) = 1/(1 + t^2)                  %
%                         ==================                  %
%                                                             %
%  So that u(1) = 1/2                                         %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  We can solve this using the Forward Euler method with      %
%  step size h, to find a numerical approximation U for u(1)  %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%  Do a series of runs with h reducing in size
%

h = 1;
j = 1;

while j < 8

h = h/2;

%
% Number of time steps
%

N = 1/h + 1;

U = zeros(1,N);
t = zeros(1,N);

U(1) = 1;
t(1) = 0;

   for i=2:N

%
%  Euler step
%
```

```
   t(i)  = h + t(i-1);

   U(i) = U(i-1) - 2*h*t(i-1)*U(i-1)^2;

   end

   hh(j) = h;
%
%  Error
%

er(j) = U(N)-1/2;

j = j+1;

end

A = [hh' er']
plot(hh,er)


-----------------------------------------------------------------------------------

%
%  The resulting solution has the following form
%

>> eul

A =

    0.5000   -0.2500
    0.2500   -0.1209
    0.1250   -0.0591
    0.0625   -0.0293
    0.0312   -0.0146
    0.0156   -0.0073
    0.0078   -0.0036

>>    h       Error
```

We can see from this that when h is halved, so is the error. The resulting solution for two values of h is given below.

6.3.2 The Runge-Kutta method.

A much more *locally* accurate method is the Runge-Kutta method. Its most famous form is called the explicit fourth order Runge-Kutta or the RK4 method. Suppose that the ODE is $\frac{\mathbf{du}}{\mathbf{dt}} = \mathbf{f}(\mathbf{t}, \mathbf{u})$. Then if

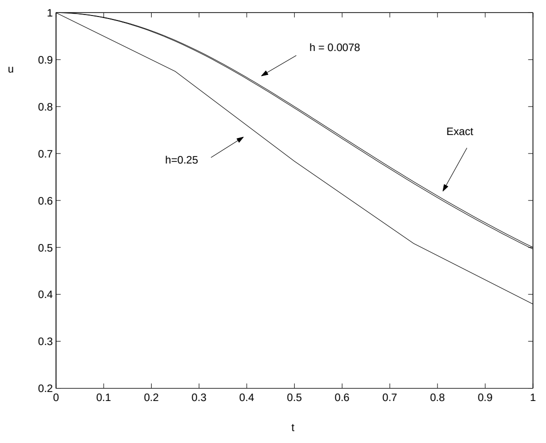


Figure 6.1: Solution of the Forward Euler Method

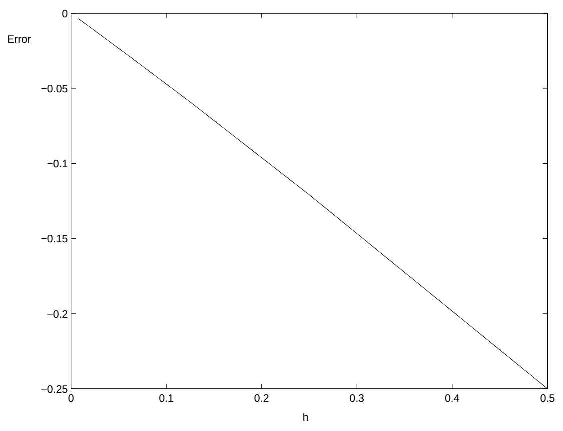


Figure 6.2: Error of the Forward Euler method as a function of h

we know $\mathbf{U}^n$, and set $t = (n-1)h$ the value of $\mathbf{U}^{n+1}$ is given by the sequence of operations

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t, \mathbf{U}^n) \\
\mathbf{k}_2 &= h\mathbf{f}\left(t + \frac{h}{2}, \quad \mathbf{U}^n + \frac{\mathbf{k}_1}{2}\right) \\
\mathbf{k}_3 &= h\mathbf{f}\left(t + \frac{h}{2}, \quad \mathbf{U}^n + \frac{\mathbf{k}_2}{2}\right) \\
\mathbf{k}_4 &= h\mathbf{f}(t + h, \quad \mathbf{U}^n + \mathbf{k}_3)
\end{aligned}$$

$$\mathbf{U}^{n+1} = \mathbf{U}_n + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)$$

It can be shown that there is a value C which depends on f in a complex way such that a local truncation error $E = |\mathbf{u}(t+h) - \mathbf{U}^{n+1}|$ is bounded by

$$E \leq Ch^5$$

Over a large number of steps these errors accumulate as before to give a global error ε of the form:

$$\varepsilon \sim C(e^{LT} - 1)h^4$$

The error is proportional to h^4. Hence the name an *order 4 method*. The error for a given h is much smaller than for the Forward Euler method. This method is *VERY* widely favoured as

1. It is easy to use and ***no*** equations need to be solved at each stage.
2. It is highly accurate for moderate h values
3. It is a *one* step method i.e. $\mathbf{U}^{n+1}$ only depends on $\mathbf{U}^n$
4. It is easy to start and easy to code.

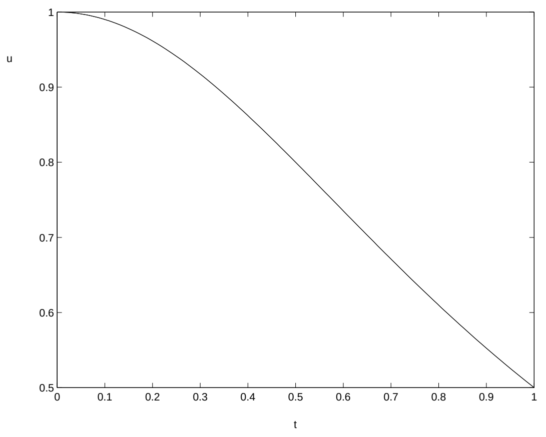
For many people it is the ONLY method they ever use!! However, it has certain disadvantages.

1. The function $\mathbf{f}$ must be evaluated four times at each iteration. This may be difficult if $\mathbf{f}$ is hard or expensive to evaluate.
2. Errors accumulate rapidly as T increases.
3. The method cannot be used for stiff problems unless h is very small.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%   Example 2: Use of the 4th Order Runge-Kutta method        %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  Consider the ODE  du/dt = -2t u^2,  u(0) = 1               %
%                    ==========================               %
%                                                             %
%  This has the solution  u(t) = 1/(1 + t^2)                  %
%                         ==================                  %
%                                                             %
%  So that u(1) = 1/2                                         %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  We can solve this using the RK4 method with                %
%  step size h, to find a numerical approximation for u(1)    %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  The function is calculated in frk.m                        %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------------%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%  Do a series of runs with h reducing in size                %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
h = 1;
j = 1;

while j < 8

h = h/2;

%
% Number of time steps
%

N = 1/h + 1;

U = zeros(1,N);
t = zeros(1,N);

%
% Initial values
%

u(1) = 1;
t(1) = 0;

   for i=2:N

%
%  Runge-Kutta step
%

   th    = h/2  + t(i-1);
   t(i)  = h    + t(i-1);


   k1    = h*frk(t(i-1),U(i-1));
   k2    = h*frk(th,U(i-1)+k1/2);
   k3    = h*frk(th,U(i-1)+k2/2);
   k4    = h*frk(t(i),U(i-1)+k3);

   U(i) = U(i-1) + (k1+2*k2+2*k3+k4)/6;

   end

   hh(j) = h;
%
%  Error
%

er(j) = U(N)-1/2;

j = j+1;

end
```

```
rat = er./hh.^4;

A = [hh' er' rat']
plot(log(hh),log(abs(er)))

>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> %
>> %  Test of the Runge-Kutta code
>> %
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> rung

A =

   0.50000000000000  -0.00029847713504  -0.00477563416071
   0.25000000000000   0.00001355253692   0.00346944945065
   0.12500000000000   0.00000139255165   0.00570389155200
   0.06250000000000   0.00000009811779   0.00643024720193
   0.03125000000000   0.00000000640084   0.00671176833566
   0.01562500000000   0.00000000040734   0.00683396495879
   0.00781250000000   0.00000000002567   0.00689065456390

>> %    h                 error             error/h^4
>> %
>> %    Note that error/h^4 is almost constant
>> %
>> diary off
```



I like to think of the RK4 method as being like a Ford Fiesta. It is easy to use, everyone uses it and it is good value for money. If you are going to the shops (i.e. solving a relatively straight forward problem) it is the method to use. However, it won't handle rough country nor would you want it for a very long drive.

To improve the accuracy the RK4 method is often used together with another *higher order* method to estimate the local error and choose h accordingly. Suppose that the value of $\mathbf{U}^{n+1}$ is given by an RK4 method. We could also use a different *higher order* method to calculate a separate value $\hat{\mathbf{U}}^{n+1}$. Both

$\mathbf{U}^{n+1}$ and $\hat{\mathbf{U}}_{n+1}$ approximate $\mathbf{u}(t+h)$ so that:

$$\begin{aligned}\mathbf{U}^{n+1} &= \mathbf{u}(t+h) + Ch^5 \\ \hat{\mathbf{U}}^{n+1} &= \mathbf{u}(t+h) + Dh^6\end{aligned}$$

Subtracting these estimates we have

$$\| \mathbf{U}^{n+1} - \hat{\mathbf{U}}^{n+1} \| \leq |C|h^5 + |D|h^6$$

If h is small then $|C|h^5 \approx \| \mathbf{U}^{n+1} - \hat{\mathbf{U}}^{n+1} \|$ so the difference between the two calculations gives an estimate for the *local* error of the RK4 method.
We can now use this estimate as follows. Given $\mathbf{U}^n$ and h

1. Using $\mathbf{U}^n$ and step size h, calculate $\mathbf{U}^{n+1}, \hat{\mathbf{U}}^{n+1}$ and $E_{n+1} = \| \mathbf{U}^{n+1} - \hat{\mathbf{U}}_{n+1} \|$.
2. *If* $\frac{\text{TOL}}{32} < E_{n+1} < \text{TOL}$ *then* accept the step
3. *If* $E_{n+1} < \frac{\text{TOL}}{32}$ *then* set $h = 2h$ and repeat from 1.
4. *If* $E_{n+1} > \text{TOL}$ *then* set $h = \frac{h}{2}$ and repeat from 1.

This method keeps the local errors below the specified tolerance and also (due to 3) makes an efficient choice of step size. However it does not control the *growth* of errors. MATLAB uses such a Runge-Kutta 4,5 pair above the form developed by Dormand & Prince in the `ode45` routine. There is a similar (cheaper but less accurate) Runge-Kutta 2,3 pair implemented in the routine `ode23`. In this routine you can set both the Absolute Tolerance (as above) or the Relative Tolerance TOL/ $\| u \|$. You can the `ode45` routine as follows:

```
>[t,U] = ode45 (@fun, trange, u_0,  options)
```

Here t is the vector of times at which the approximate solution vector U is given. Because the step-size h is chosen at each stage of the algorithm these times will not (necessarily) be equally spaced. The function $f(t,u)$ is specified by 'fun' and u_0 is the vector of initial conditions. The time interval for the calculation is given by trange. Setting

```
trange = [a b]
```

means that the ode45 routine will integrate the ODE between a and b, giving its output at the (variable) time steps it computes. Alternatively

```
trange = [a: c: b]
```

leads to output at the points $a, a+c, a+2c$ etc. The options command is optional, but it allows control over the operation of the ode45 routine. In particular you can set the tolerances and/or request statistics on the solution. The options are set by using the odeset routine. For example if you want an absolute tolerance of 10^{-12} you type

```
>options = odeset ('AbsTol', 1e^(-12}))
```

By default the absolute tolerance is 10^{-6} and the relative tolerance is 10^{-3}.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                              %
%   Example 3: Use of ode45                                    %
%                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                              %
%  Consider the ODE  du/dt = -2t u^2,  u(0) = 1                %
%                    ==========================                %
%                                                              %
%  This has the solution  u(t) = 1/(1 + t^2)                   %
%                         ==================                   %
%                                                              %
%  So that u(1) = 1/2                                          %
%                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                              %
%  We can solve this using the ode45 method with               %
%  tolerances 1e-12,to find a numerical approximation for u(1)%
%                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


options=odeset('AbsTol',1e-12,'RelTol',1e-12);

trange = [0:0.05:1];
unit   = [1];

[t,U]  = ode45(@frk,trange,unit,options);

s = size(U);

err = U(s(1)) - 0.5
```

6.3.3 The Störmer - Verlet method and geometric integration

Geometric integration is a branch of numerical analysis which aims in part to control *global* error growth of a numerical approximation over long times. An important application of geometric methods is to systems of the form

$$\left.\begin{array}{c}\dot{u} = v \\ \dot{v} = -f(u)\end{array}\right\} \{ \ \ddot{u} + f(u) = 0.$$

More generally, geometric methods can be applied to Hamiltonian systems for which

$$\dot{q} = \partial H/\partial p, \quad \dot{p} = -\partial H/\partial q$$

For the problem $\ddot{u} + f(u) = 0$ we set $q = u, p = du/dt$ and

$$H = p^2/2 + F(q) \quad \text{with} \quad F = \int f dq$$

Multiplying the differential equation by $\dot{u}$ and integrating with respect to time we find that

$$H = \dot{u}^2/2 + F(u) = \text{const.}$$

More generally, in an autonomous Hamiltonian system, the Hamiltonian H is a *constant* for all times. This is an example of a conservation law. Many physical systems conserve particular quantities over all times. An excellent example of this is the solar system considered in isolation with the rest of the universe. This obeys a complicated set of differential equations with complex (and indeed chaotic) solutions. However the total energy, angular momentum and linear momentum are conserved for *all* time.

To retain the correct dynamics of such a system in a numerical approximation it is essential that the numerical approximation either exactly conserves the same invariants or (more usually) the approximate equivalent of any conserved quantity varies from a constant by a small but *bounded* amount for *all* times.

If this occurs then the approximate solution is likely to be much closer to the true solution for all times.

The Forward Euler method, RK4 and ode45 are not good at preserving such conserved quantities. For example if we consider the system

$$\ddot{u} + f(u) = 0, \quad \frac{\dot{u}^2}{2} + F(u) = H$$

An application of the Forward Euler method with $U^n \approx u((n-1)h), V^n \approx \dot{u}((n-1)h)$ gives

$$U^{n+1} = U^n + hV^n,\ V^{n+1} = V^n - hf(U^n)$$

so that

$$\begin{aligned} H_{n+1} &= \frac{(V^{n+1})^2}{2} + F(U^{n+1}) \\ &= \left[V^n - hf(U^n)\right]^2/2 + F(U^n + hV^n) \\ &= H_n + \frac{h^2}{2}\left(f(U^n)^2 + (V^n)^2\partial f/\partial u\right) + O(h^3) \end{aligned}$$

Thus H changes by

$$H_{n+1} - H_n = \frac{h^2}{2}\left(f(U^n)^2 + (V^n)^2\partial f/\partial u\right) + O(h^3)$$

at each iteration. If $\partial f/\partial u > 0$ then H is not conserved and increases at each iteration as the errors accumulate so that the global error in H grows like nh^2. In the RK4 method we have the better result that

$$H_{n+1} - H_n = h^5 C_n.$$

Again, in general the values of C_n are positive for many problems and the errors in H accumulate over a large number of iterations, with global errors growing like nh^5. A widely used method which avoids many of these problems and has excellent conservation problems is the Störmer - Verlet method (SV) . This is <u>the</u> method of choice for simulations of celestial and molecular dynamics. Not only is it (much) better than RK4 for long-time integrations it is also much cheaper and easier to code up as it only requires one function evaluation per time step. It is also explicit, has global errors $O(h^2)$ and it is symmetric. The disadvantage of the SV method is that it can only be used for a certain class of problems (those with a separable Hamiltonian) and it is not a black-box code i.e. it requires some thought to use it. However this is precisely what mathematicians are paid to do!
For the problem $\ddot{u} + f(u) = 0$ the SV method takes the following form

$$\begin{aligned} U^* &= U^n + \tfrac{h}{2}V^n \\ V^{n+1} &= V^n - hf(U^*) \\ U^{n+1} &= U^* + \tfrac{h}{2}V^{n+1} \end{aligned}$$

In the SV method we have

$$H_{n+1} - H_n = h^3 D_n.$$

This appears to be worse than RK4. However, unlike RK4 the errors DO NOT accumulate and tend to cancel out. It can be shown that if H is the exact Hamiltonian then

$$|H_n - H| < Dh^3$$

where D does not depend on n. So, although H is not exactly conserved, the method stays close to it for all time.
We now apply this to an example with $f(u) = u^3$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                %
%  Example 4: Stormer-Verlet method for          %
%                                                %
%  u'' + u^3 = 0, u(0) = 1, u'(0) = 0            %
%                                                %
%  In the exact eqn  H is constant where         %
%                                                %
%  H = (u')^2/2 + u^4/4                          %
%                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

t(1) = 0;
U(1) = 1;
V(1) = 0;
h    = 0.1;
H(1) = 1/4;

   for i=2:100

   Usta = U(i-1) + (h/2)*V(i-1);
   V(i) = V(i-1) - h*Usta^3;
   U(i) = Usta   + (h/2)*V(i);

   t(i) = t(i-1) + h;
   H(i) = V(i)^2/2 + U(i)^4/4;

   end

plot(t,H)
```

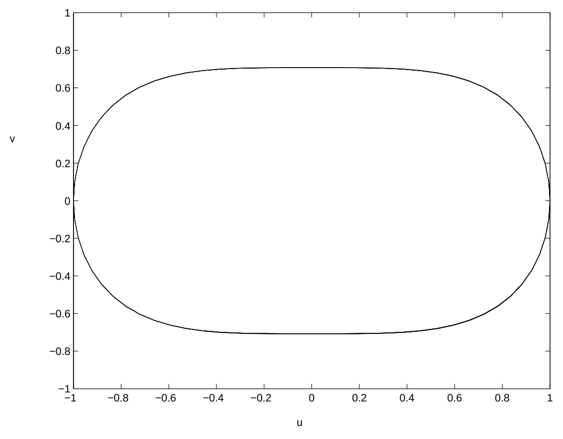


Figure 6.3: A plot of U, V for the exact and numerical solution

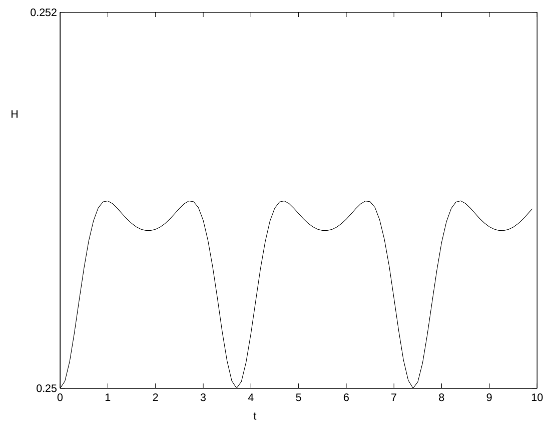


Figure 6.4: A plot of H showing the bounded variation

6.4 Stiff Differential Equations

6.4.1 Definition

In a *stiff* differential equation the solution components evolve on very different timescales. This causes a problem as a numerical method, such as ode45, possibly chooses a step size for the most rapidly evolving component, even if its contribution to the solution is negligable. This leads to very small step sizes, highly inefficient computations and long waits for the user! The reason for this is an *instability* in the method , where a small error may grow rapidly with each step.

Suppose we want to solve the ODE

$$\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u})$$

and the numerical method makes a small error $\mathbf{e}$. We ask the question, how does e grow during the calculation? To answer this we start by looking at how a small disturbance to the solution of the ODE changes. Suppose that $\mathbf{e}$ is such a disturbance so that

$$\frac{d}{dt}(\mathbf{u} + \mathbf{e}) = \mathbf{f}(\mathbf{u} + \mathbf{e})$$

To leading order the perturbation $\mathbf{e}$ then satisfies the linear differential equation

$$\frac{d\mathbf{u}}{dt} + \frac{d\mathbf{e}}{dt} = \mathbf{f}(\mathbf{u}) + \mathbf{Ae} \qquad A = \frac{\partial \mathbf{f}}{\partial \mathbf{u}}.$$

The perturbation growth is thus described by the differential equation

$$\frac{d\mathbf{e}}{dt} = A\mathbf{e} \quad \text{where} \quad A = \partial \mathbf{f}/\partial \mathbf{u}. \tag{6.8}$$

Now look at the ODE (6.8). This has the solution

$$\mathbf{e} = e^{At}\mathbf{e_0}$$

where $\mathbf{e}_0$ is the initial perturbation. So that if $A = U\Lambda U^{-1}$ then

$$\mathbf{e} = Ue^{\wedge t}U^{-1}\mathbf{e}_o.$$

Alternatively, if A has eigenvalues λ_i and eigenvectors ϕ_i then

$$A\phi_i = \lambda_i\phi_i \qquad \phi_i : \text{eigenvector}$$

Thus if $\mathbf{e}_0$ is in the direction of ϕ_i so that

$$\mathbf{e}_0 = a\phi i$$

it follows that

$$\mathbf{e}(t) = ae^{\lambda_i t}\phi_i$$

It follows further that

$$\parallel \mathbf{e}(t) \parallel = |a|e^{\mu_i t} \parallel \phi_i \parallel$$

where μ_i is the *real part* of λ_i so that it is the real part of the eigenvalues which control the growth of small perturbations.
If $|\lambda_i|$ is large and the real part of λ_i is less than zero, then the contribution to $\mathbf{e}$ in the direction of the eigenvector ϕ_i rapidly decays to zero. After only a short time the perturbation is dominated by the component in the direction of the eigenvector ϕ_j for which λ_j has the largest *real* part over all the eigenvalues.
We are now able to define what we mean by a stiff system.

DEFINITION
The system is *stiff* if the matrix A has eigenvalues λ_i for which

$$\max_j |\lambda_j| \gg \min_j |\lambda_j|.$$

Typically, in an application a ratio of over 10 is considered to lead to a stiff system

In a physical system the components for which $|\lambda_j|$ is large, and the real part of λ_j is negative, decay rapidly and are not seen in the solution apart from some initial transients. It is therefore somewhat paradoxical that it is *precisely* these components which lead to instabilities in the numerical scheme.This is another way of saying that the solution has components that evolve at very different rates.

6.4.2 Numerical Methods

Now we look at a variety of numerical methods for solving the linear equation (6.8) so that we may compare the growth of perturbations to the numerical solution with those of the true solution. First look at the performance of the *Forward Euler* method when applied to this problem. We have

$$\mathbf{U_{n+1}} = \mathbf{U_n} + hf(\mathbf{U_n})$$

so that

$$\mathbf{U_{n+1}} = \mathbf{U_n} + hA\mathbf{U_n} = (I + hA)\,\mathbf{U_n}$$

Therefore

$$\mathbf{U_n} = (I + hA)^n\, U_1$$

But $I + hA$ has the same eigenvectors ϕ_j as A and has eigenvalues

$$1 + h\lambda_j$$

So, the contribution to $\mathbf{U}_n$ in the direction of the eigenvector ϕ_j grows as

$$(1 + h\lambda_j)^{n-1}$$

or more precisely as $|1 + h\lambda_j|^{n-1}$. In particular the errors caused by truncation error or by rounding error only *decay* if $|1 + h\lambda_j| < 1$ for all λ_j.

We now find an extraordinary paradox. Suppose that λ_j is real and that

$$\lambda_j = -\mu \quad \text{with} \quad \mu \gg 1$$

Then

$$e^{\lambda_j t} = e^{-\mu t} \ll 1, \quad \text{if} \quad t \gg 1$$

However

$$|1 + h\lambda_j|^n = |1 - \mu h|^n \gg 1 \quad \text{if} \quad \mu h > 2 \quad \text{and} \quad n \gg 1.$$

So the most rapidly decaying components of the perturbation to the continuous solution are the components of $\mathbf{U}_n$ which are growing most rapidly.

DEFINITION

We say the Forward Euler method with step size h is *stable* if given a matrix A with eigenvalues λ_j with the real part less than zero *then* $|1 + h\lambda_j| < 1$ for all λ_j. In particular if λ_j are all real then the method is stable only if

$$h < 2/\max|\lambda_j|$$

This is the restriction on h which means that solution errors do not grow. If h is larger than this bound then errors grow and the method is unstable.

Here we see the problem. A component in the direction of ϕ_j with large $|\lambda_j|$ and with real part $\lambda_j < 0$ dominates the choice of step size even though this component is very small.

- The *LOCAL (TRUNCATION) ERROR* that is made at each stage of the calculation is given by

$$\frac{h^2|\mathbf{u}''|}{2}.$$

  So the error made at *each* stage depends on the *SOLUTION.* If all of the eigenvalues of A have negative real part then this error is ultimately dominated by the component of the solution that decays most slowly, which is in turn determinated by the eigenvalue of A with the *largest real part.* Soi that it is the eigenvalue with the real part closest to zero, and typically this is the eigenvalue with the *smallest* modulus.

- In contrast the *GROWTH* of the error as the iteration proceeds depends on the eigenvalue of A with the largest modulus, even though this eigenvalue may have a large negative real part and thus not contribute to the solution in any way.

There are therefore two restrictions on h, it must be small both for *accuracy* at each stage and for *stability* to stop the errors growing. Stiffness arises when the restriction on h for stability is much more severe than the restriction for accuracy.

This introduces us to the ideas of stability and instability. It is surprisingly hard to give a precise definition of what we mean by instabiliy in a numerical method which accounts for all cases - later on we will give a precise definition which covers certain cases, but for the present we will have the following.

INFORMAL DEFINITION OF INSTABILITY
A numerical method to solve a differential equation is *unstable* if the errors it makes (or indeed its solution) grow more rapidly than the underlying solution. If the errors decay then it is stable.

Exercise Check the definition of stability for the Forward Euler method is consistent with this informal definition.

Returning to the Forward Euler example. If $h > 0$ and $\lambda = p + iq$ then $|1 + h\lambda| < 1$ implies that

$$(1 + hp)^2 + (hq)^2 < 1$$

$$2hp + h^2p^2 + h^2q^2 < 0$$

$$2p + hp^2 + hq^2 < 0.$$

So that the method is stable if (p, q) lies in the circle of radius $\frac{1}{h}$ shown in Figure 6.5. In this figure

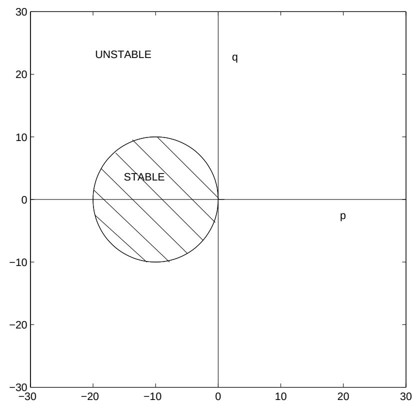


Figure 6.5: Stability region for the Forward Euler method

the shaded region shows the values of p and q for which the numerical method is stable. Recall that the original differential equation is stable provided that p lies in the half-plane $p < 0$. The shaded region only occupies a fraction of this half-plane, although the size of the shaded region increases as $h \to 0$. Thus, for a *fixed* value of h the numerical method will only have errors which do not grow if the eigenvalues of A are severely constrained. Unfortunately this is often not the case, particularly in discretisations of partial differential equations.

6.4.3 The Backward-Euler Method... an A-stable stiff solver

We now look at another method, *The Backward Euler Method* (also called the BDF1 method). This is given by

$$\mathbf{U}_{n+1} = \mathbf{U_n} + h\mathbf{f}(\mathbf{U}_{n+1})$$

The backward Euler method is **much** harder to use than Forward Euler as we must solve an equation (which is usually nonlinear) at *each* step of the calculation to find $\mathbf{U_{n+1}}$. It has the same order of error. i.e. the global error is proportional to h. If we now apply this to the equation $\dot{u} = Au$ we have

$$\mathbf{U_{n+1}} = \mathbf{U_n} + hA\mathbf{U_{n+1}}$$

This is a linear system which we need to invert to give

$$\mathbf{U_{n+1}} = (I - hA)^{-1}\mathbf{U_n}$$

Now, if the eigenvalues of A are λ_j, those of $(I - hA)^{-1}$ are $(1 - h\lambda_j)^{-1}$, with the same eigenvectors.

Exercise: Prove this.

Thus the contribution to $\mathbf{U_n}$ in the direction of ϕ_j evolves as

$$|1 - h\lambda_j|^{1-n}$$

Now let $\lambda_j = p + iq$ as before. It follows that

$$|1 - h\lambda_j|^{-2} = \frac{1}{(1 - hp)^2 + (hq)^2}$$

Therefore the errors decay and the method is stable if

$$\frac{1}{(1 - hp)^2 + (hq)^2} < 1$$

$$\text{if} \quad 1 < (1 - hp)^2 + (hq)^2$$

$$\text{or} \quad 0 < h(p^2 + q^2) - 2p$$

The resulting stability region is illustrated (shaded) in Figure 6.6: This picture is in complete contrast to the one that we obtained for the Forward Euler method. The stability region is now very large and certainly includes the half-plane $p < 0$. Thus any errors in the numerical method will be rapidly damped out. Unfortunately the numerical solution can decay even if $p \geq 0$, so that neutral or growing terms in the underlying solution can be damped out as well. This is a source of (potential) long term error, especially in Hamiltonian problems.

The Backward Euler method is very reliable and has other nice properties (a maximum principle) which make it especially suitable for solving PDEs. The main disadvantage to using it is that we must solve a nonlinear equation to find $\mathbf{U_{n+1}}$. This is an example of the basic principle that *there is no such thing as a free lunch!* The penalty of a stable method is the need to do more work. Note, however, that the Störmer - Verlet method is a good approximation to a free lunch. Finding $\mathbf{U_{n+1}}$ when the system has a high dimension (e.g.10^4) is a considerable task, especially as this calculation must be done quickly and often. Usually we use an iterative method such as the Newton-Raphson or Broyden method. Fortunately a good initial guess is available namely a value $\hat{\mathbf{U}}_{\mathbf{n+1}}$ which is obtained by taking *one-step* of an explicit method (e.g Forward Euler) applied to $\mathbf{U}_n$. This procedure is called a *predictor-corrector* method in which the explicit method predicts the value $\hat{\mathbf{U}}_{\mathbf{n+1}}$ which is then corrected (usually by an iterative method) to give $\mathbf{U_{n+1}}$.

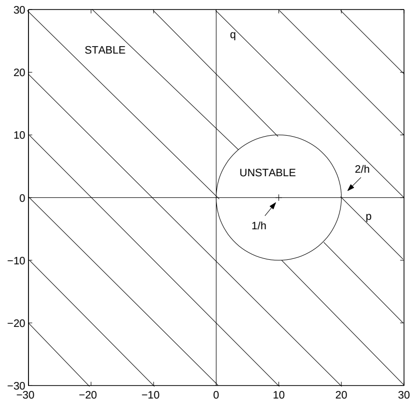


Figure 6.6: Stability region for the Backward Euler method.

6.4.4 The Trapezium Rule: a symmetric stiff solver

The Trapezium Rule is given by

$$\mathbf{U_{n+1}} = \mathbf{U_n} + \frac{h}{2}\left[\mathbf{f}(\mathbf{U_n}) + \mathbf{f}(\mathbf{U_{n+1}})\right]$$

This is another *implicit* method which needs a function solve at each step to find $\mathbf{U}_{n+1}$ using a predictor-corrector method.
It is also a *symmetric* method i.e. if you know $\mathbf{U_n}$ and you wish to find $\mathbf{U_{n+1}}$ with step-size h then this is the same method for finding $\mathbf{U_n}$ given $\mathbf{U_{n+1}}$ and step-size $-h$. This property is important for finding approximations to the solutions equations such as $\ddot{u} + u = 0$ which are the *same* both forwards and backwards in time.

If we now apply this method to $\dot{\mathbf{u}} = A\mathbf{u}$ we have

$$\mathbf{U_{n+1}} = \mathbf{U_n} + \frac{h}{2}\left[A\mathbf{U_n} + A\mathbf{U_{n+1}}\right]$$

so that

$$\mathbf{U_{n+1}} = \left(I - \frac{hA}{2}\right)^{-1}\left(I + \frac{hA}{2}\right)\mathbf{U_n}$$

A straight forward calculation of the eigenvalues of the matrix linking U_n to U_{n+1} shows that growth rate of the component in the direction of ϕ_j is given by

$$\left|\frac{1 + \frac{h\lambda_j}{2}}{1 - \frac{h\lambda_j}{2}}\right|$$

If we take $\lambda = p + iq$ then

$$\theta \equiv \left|\frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}\right|^2 = \frac{\left(1 + \frac{ph}{2}\right)^2 + q^2 h^{\frac{2}{4}}}{\left(1 - \frac{ph}{2}\right)^2 + q^2 h^{\frac{2}{4}}}$$

Thus

$$\theta < 1 \quad \text{if} \quad \left(1 + \frac{ph}{2}\right)^2 + q^2 < \left(1 - \frac{ph}{2}\right)^2 + q^2 \quad \text{i.e if} \quad p < 0$$

The stability region of the numerical method is thus the half-plane $p < 0$ illustrated in Figure 6.7 which is *identical* to the region of stability of the underlying differential equation. As a consequence the dynamics of the solution of the trapezium rule exactly mirrors the true dynamics. This is an excellent state of affairs.

The local error LTE of the Trapezium Rule is given by:

$$LTE = \frac{h^3|\mathbf{u}'''}{12}.$$

If h is constant the overall error is then proportional to h^2. To estimate the step-size we keep $LTE < TOL$ much as before.

The Trapezium Rule, together with a third order method to control the local error, is implemented in the MATLAB routine `ode23t` . Here the (second order implicit) Trapezium Rule is a good one to use if accuracy is not essential. It is very reliable and relatively cheap, but the overall error is quite high compared to (say) `ode45`.

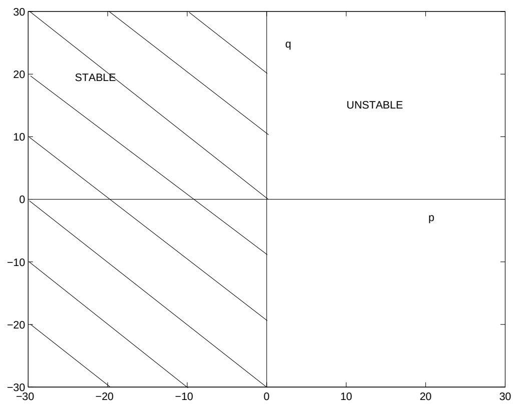


Figure 6.7: Stability Region for the Trapezium rule and the Implicit Mid-Point rule.

6.4.5 The Implicit Mid-point Rule...an ideal stiff solver?

The implicit mid-point rule is a symmetric Runge-Kutta method closely related to the trapezium rule. It is given by

$$\mathbf{U_{n+1}} = \mathbf{U_n} + h\mathbf{f}\left(\frac{1}{2}(\mathbf{U}_n + \mathbf{U}_{n+1})\right) \tag{6.9}$$

When applied to the linear ODE $\mathbf{u}' = Au$ the method in (6.9) gives exactly the same sequence of iterates as the trapezium rule *check this*. Thus its stability properties are identical to the Trapezium Rule and hence are optimal. Like the Trapezium Rule the global error of the Implicit Mid-Point Rule varies as h^2 and a function solve is required to find $\mathbf{U_{n+1}}$. It has various other very nice features. In particular it preserves linear and quadratic invariants, so that if $\mathbf{u}$ is a solution of the differential equation and there exists a vector $\mathbf{c}$ and a matrix A so that $\mathbf{c} \cdot \mathbf{u}$ is constant and $\mathbf{u}^T A\mathbf{u}$ is a constant then the same identities hold for the discrete solution as well. It is also a symplectic method (like the Störmer- Verlet method). An example of the usefulness of these properties comes from the computation of the orbits of the planets in the solar system. A linear invariant of this system is the linear momentum and a quadratic invariant is the angular momentum. Both are *exactly* conserved by this method, which also comes close to conserving the total energy. Great stuff, but at the cost of an expensive function solve.

The Implicit Mid-point rule is the simplest example of a sequence of implicit Runge-Kutta methods called Gauss-Legendre methods. If you want to solve an ODE very accurately for long times with excellent stability, but regardless of cost, then these are the methods to use. Gauss-Legendre methods are the Lamborghinis of the numerical ODE world.They are expensive and hard to drive, but they have style! They are implemented in Fortran in the excellent AUTO code.

6.4.6 Multi-step methods

These are the most widely used methods for stiff problems and include Adams and BDF methods and the Trapezium rule. There is a vast literature on them, see Iserles. Multi-step methods are very flexible to use and are relatively easy to analyse. They take the form

$$\sum_{l=0}^{k} a_l \, \mathbf{U}_{n-l} = h \sum_{l=0}^{k} b_l \, \mathbf{f}_{n-l}$$

where $\mathbf{f}_k = \mathbf{f}(t_k, \mathbf{U}_k), \quad t_k = (k-1)h.$
To find $\mathbf{U}_n$ you need to know $\mathbf{U}_{n-1}, \ldots, \mathbf{U}_{n-k}$. To *start* the method given $\mathbf{U}_0$ you need to use another method (e.g. RK4) to find $\mathbf{U}_1, \ldots \mathbf{U}_{k-1}$.

In these methods you use information from previous time steps and (in an explicit method) one function evaluation to find $\mathbf{U}_n$. Thus they are cheaper than RK4 and potentially more accurate. Some properties of these methods are as follows.

- The method has order if the *truncation error* at each stage is given by $Ch^{p+1}u^{(p+1)}$ and the overall method has error of order p, so that the global error is proportional to h^p.
- These methods are *implicit* if $b_o \neq 0$ and *explicit* if $b_0 = 0$. Explicit methods give $\mathbf{U_n}$ directly whereas implicit methods need an equation to be solved.
- They have "backwards difference form" if $b_o \neq 0, \; b_l = 0$ otherwise

DEFINITION
A multi-step method is *A-stable* if when used to solve the equation

$$\mathbf{u}' = A\mathbf{u},$$

when all eigenvalues of A have negative real part, then $|\mathbf{U}_n| \to 0$ as $n \to \infty$, for all values of $h > 0$. More precisely, a method is A-stable if the roots z of the polynomial equation $\Sigma a_l z^{-l} - h\lambda \Sigma b_l z^{-l} = 0$ have modulus less than or equal to one if the real part of λ is less than or equal to zero.

A-stability is a very desirable property for stiff problems, which the Trapezium Rule has. However, it is almost unique in having this property as the following theorem shows:

THEOREM Only *implicit* methods of order less than or equal to 2 can be A-stable.

Implicit Multi-step methods involve solving non-linear equations. As before the usual method to do this is a predictor corrector method namely you generate an approximation $\hat{\mathbf{U}}$ for $\mathbf{U}_n$ using an *explicit predictor* method and then correct this to find U_n. An easy corrector is to use an iterative one. Suppose we set $\mathbf{U}_n^o = \hat{\mathbf{U}}_n$ where $\hat{\mathbf{U}}_n$ is a predicted value obtained using an explicit method. We now perform the following iteration to find a sequence of approximations $\mathbf{U}_n^r$ to $\mathbf{U_n}$:

$$\mathbf{U}_n^r = \left[-\sum_{l=1}^{k} a_l \mathbf{U}_{n-l}^{r-1} + h \sum b_l \tilde{f}_{n-l} \right] / a_o$$

where

$$\tilde{f}_k = f(t_k, \mathbf{U}_k^{r-1}),$$

iterating either to convergence or a fixed number of times.

As in the Runge-Kutta methods, it is common to use two multistep methods simultaneously. *One* to perform the numerical solution. The *other* to give an estimate of the error. In the celebrated Gear solvers (named after their inventor W.Gear) the method chooses what multi-step method to use at each step from a range of methods of different orders (and stability ranges), based on an estimate of the error from the two methods (the latter is called the Milne device).

MATLAB has an excellent stiff Gear solver given by `ode15s` . It is used in exactly the same way as `ode45` and is the routine to use for most problems, if you don't know much about their structure.

***IF YOU LEARN NOTHING ELSE FROM THIS CHAPTER IT IS TO USE ODE15S

An important sub-set of multi-step methods are BDF (backwards difference form) methods. BDFn methods have order n (global errors proportional to h^n) and BDF1 is just the Backwards Error method. The important Fortran ODE code DDASSL and `ode15s` are both based on BDF methods and their extensions. The first three BDF methods are given by:

$$\begin{array}{ll} \text{BDF1 :} & \mathbf{U}_n - \mathbf{U}_{n-1} = h\,\mathbf{f}_n \qquad \text{[Backward Euler]} \\ \text{BDF2 :} & \mathbf{U}_n - \frac{4}{3}\mathbf{U}_{n-1} + \frac{1}{3}\mathbf{U}_{n-2} = \frac{2}{3}h\mathbf{f}_n \\ \text{BDF3 :} & \mathbf{U}_n - \frac{18}{11}\mathbf{U}_{n-1} + \frac{9}{11}\mathbf{U}_{n-2} - \frac{2}{11}\mathbf{U}_{n-3} = \frac{6}{11}h\,\mathbf{f}_n \end{array}$$

These methods have excellent stability properties: BDF2 is A-stable (damping out errors but not being too dissipative). BDF3 is $A(\alpha)$ stable rather than A-stable; its stability region includes a wedge of angle α and this includes the eigenvalues of many problems such as those arising in fluid mechanics.
BDF methods are the Land Rovers of ODE solvers. They may not be pretty or easy to use, but they will handle rough country and will nearly always get you where you want to go (although they are not to be used for Hamiltonian problems!).

6.5 Differential Algebraic Equations (DAEs)

An important application of BDF methods is to *differential algebraic equations*. These equations combine algebraic and differential equations and they are VERY common in many applications for example chemistry, electronics, fluid mechanics and robotics. In a DAE we think of $\mathbf{U} = \begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix}$ as satisfying the equations

$$\dot{\mathbf{p}} = \mathbf{f}\,(\mathbf{p},\mathbf{q}), \quad \mathbf{0} = \mathbf{g}\,(\mathbf{p},\mathbf{q}). \tag{6.10}$$

The second of these equations is the algebraic (or constraint) equation.We will first look at the scalar case. If we differentiate this with respect to t we have

$$0 = \frac{\partial g}{\partial p}\dot{p} + \frac{\partial g}{\partial q}\dot{q}.$$

If dg/dq is invertible we then have

$$\dot{q} = -\left(\frac{\partial g}{\partial q}\right)^{-1}\frac{\partial g}{\partial p} \quad \dot{p} = -\left(\frac{\partial g}{\partial q}\right)^{-1}\frac{\partial g}{\partial p}f$$

Thus we arrive at a differential equation for q. We call problems of this form *index-1* problems. If we have to differentiate *twice* with respect to t to get a differential equation for q we call this an *index-2* problem, and if three differentiations are needed an *index-3* problem. Electronics and chemistry tend to lead to index-1 problems, fluid mechanics to index-2 problems and robotics to index-3 problems. MATLAB can handle index-1 problems but has difficulties with problems of a higher index. Full details on the theory and computation of DAEs are given in the very readable book by U. Ascher and L. Petzold.

It is not at all obvious how to apply an explicit method to solve such equations, especially to ensure that the constraint $\mathbf{g}(\mathbf{p}, \mathbf{q}) = \mathbf{0}$ is met at each stage.
However, it is easy to code these up using a BDF method. For example if we apply BDF2 to the DAE (6.10) we get:

$$\boxed{\begin{array}{llll} \mathbf{p}_n - \frac{4}{3}\,\mathbf{p}_{n-1} + \frac{1}{3} & \mathbf{p}_{n-2} & = \frac{2}{3}h & \mathbf{f}(\mathbf{p}_n, \mathbf{q}_n) \\ & \mathbf{0} & = & \mathbf{g}(\mathbf{p}_n, \mathbf{q}_n) \end{array}}$$

As before we have to solve Nonlinear equations to find p_n and q_n, but these equations are no worse to solve than before. In *ddassl* these equations are solved using *quasi-Newton* methods in which *conjugate-gradient* methods are used to speed up the linear algebra.

The code `ode15s` can solve DAEs of the form

$$M\dot{u} = f(u)$$

Here the matrix M can be singular, for example if

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, f \equiv \begin{pmatrix} f \\ g \end{pmatrix} \quad \text{and} \quad u \equiv \begin{pmatrix} p \\ q \end{pmatrix}$$

then we have $\dot{p} = f$ and $0 = g$ as in (6.10).
When using `ode15s` in this way you specify M in advance and then proceed much as before. More details are given by help ode15s.

Chapter 7

Two Point Boundary Value Problems (BVPs)

7.1 Introduction

Two point boundary problems (2pt BVPs) take the form:

$$\boxed{\begin{array}{lll} \dot{\mathbf{u}} & = \mathbf{f}(x, \mathbf{u}) & \mathbf{u} \in \mathbb{R}^n \\ \mathbf{g}_1(\mathbf{u}(a)) & = \boldsymbol{\alpha} & x \in \mathbb{R} \\ \mathbf{g}_2(\mathbf{u}(b)) & = \boldsymbol{\beta} & \end{array}}$$

where $\mathbf{g}_1, \boldsymbol{\alpha} \in \mathbb{R}^m$ and $\mathbf{g}_2, \boldsymbol{\beta} \in \mathbb{R}^{n-m}$. Here x is usually thought of as a *spatial* variable.
Two point BVPs arise in many contexts of which the following are examples.

1. They are one-dimensional elliptic equations and arise in their own right as descriptions of physical problems. For example, the equation for the deflection of the Euler strut is given by:

   $$d^2u/dx^2 + \lambda \sin(u) = 0 \quad u(0) = u(1) = 0,$$

   and the equation for rock deformation (see work by CJB and Professor Giles Hunt) by:

   $$d^4u/dx^4 + Pd^2u/dx^2 + f(u) = 0, \quad u(0) = u''(0) = u(1) = u''(1) = 0$$

2. As *steady states* of *parabolic* or hyperbolic partial differential equations. For example the limit of the solutions of

   $$\partial u/\partial t = \partial u/\partial x - \mathbf{f}(x, u)$$

   in the limit of $t \to \infty$ [Often a good way to solve the steady state problem is to convert it to such a time-dependent problem].

3. As *travelling wave solutions* of partial differential equations or (more generally) as *similarity reductions* reductions of partial differential equations. (These will be described in the semester Two course on Mathematical Modelling.) An example of this is given by the Bergers' equation:

   $$\frac{\partial \theta}{\partial t} + \theta \frac{\partial \theta}{\partial z} = \epsilon \frac{\partial^2 \theta}{\partial z^2}$$

   If we look for a travelling wave solution with

   $$\theta(z, t) = u(x) \quad \text{where} \quad x = z - ct$$

and

$$\theta(-\infty,t)=1, \quad \theta(\infty,t)=-1.$$

Then u satisfies the BVP

$$c\frac{du}{dx}+u\frac{du}{dx}=\epsilon\frac{d^2u}{dx^2}$$
$$u(-\infty)=1, \quad u(\infty)=-1.$$

which is a two-point boundary value problem.

A special (and very hard) case of two-point BVPs is given when $a=-\infty$ and/or $b=\infty$. Here we look for *ground* state, homoclinic or heteroclinic solutions. These are very common in PDEs as *self-similar* solutions or in quantum mechanics as *bound-states* We have just seen such a problem in the travelling wave example.

There is a *huge* difference between BVPs and IVPs. In general an initial value problem always has a unique solution. However a BVP may have *one, several* (sometimes ∞) or *no* solutions.
The theory of such problems is very subtle. An excellent description of this and of numerical methods for them is given in the book on 2pt BVPs by Ascher, Matheij and Russell. There are many methods for solving BVPs. These include shooting methods, finite difference methods, finite element methods, spectral methods and collocation methods. The latter are used in the MATLAB code `bvp4c` and in Fortran codes such as COLSYS, MOVCOL and AUTO. They are especially suitable for use with adaptive and non-uniform meshes. In this course we will look at shooting methods and finite difference methods.

7.2 Shooting Methods

These are the simplest methods for solving BVPs and are based on using IVP software such as ode15s. They are easy to use and it is often worth trying them first regardless of the nature of the underlying problem, although they can perform badly on problems with boundary laters. In a shooting method you reduce the problem to one of finding the correct boundary values. As this is such a neat idea you should:

SHOOT FIRST AND ASK QUESTIONS LATER!

The idea behind shooting methods is simple. Let $\mathbf{y}$ be a solution of the initial value problem

$$\frac{d\mathbf{y}}{dx}=\mathbf{f}(x,\mathbf{y}), \quad \mathbf{y}(a)=\mathbf{z}\in I\!R^n \tag{7.1}$$

Suppose we take general initial conditions $\mathbf{y}(a)=\mathbf{z}$. The initial value problem (7.1) can always be solved for such general values so that $\mathbf{y}(b)$ is then a function $\mathbf{F}(\mathbf{z})$ of $\mathbf{z}$. If we can find a suitable vector $\mathbf{z}$ so that

$$\mathbf{g}_1(\mathbf{z})=\boldsymbol{\alpha} \quad \text{and also} \quad \mathbf{g}_2(\mathbf{F}(\mathbf{z}))=\boldsymbol{\beta}$$

then we will have found a solution of the corresponding boundary value problem by simply setting $\mathbf{u}(x)\equiv\mathbf{y}(x)$.

The procedure works as follows

1. At the point $x=a$ we have $\mathbf{g}_1(\mathbf{z})=\boldsymbol{\alpha}$. As $\mathbf{g_1}\in I\!R^m$ this condition fixes m components of the vector $\mathbf{z}$ say $z_1,\ldots,z_m$. The other $n-m$ components $(z_{m+1},\ldots,z_n)$ can be chosen freely.
2. Next use an initial value solver such as `ode15s` to calculate $\mathbf{y}(b)\equiv\mathbf{F}(\mathbf{z})$.

3. To satisfy the boundary conditions we must have $\mathbf{g}_2(\mathbf{F}(\mathbf{z})) = \boldsymbol{\beta}$. However, as $\mathbf{g}_2 \in I\!\!R^{n-m}$ we have $n-m$ (usually nonlinear) equations to be satisfied for the $n-m$ unknowns $z_{m+1}, \ldots, z_n$.
   These nonlinear equations can be solved using a nonlinear solver such as the Newton-Raphson algorithm or a MATLAB routine such as `fzero` or `fminsearch`.

Shooting methods perform *well* for problems without boundary or internal layers. For those with boundary layers the effects of exponentially growing terms make these methods hopelessly ill conditioned and unusable. There are extensions of shooting methods called *multi-shooting methods* which can be used for problems with boundary layers. However these are very problem dependent and are not easy to use.

Example 1

The easiest example of an application of these methods is to linear boundary value problems of the form

$$a(x)\frac{d^2u}{dx^2} + b(x)\frac{du}{dx} + c(x)u = d(x)$$

with the boundary conditions $u(0) = 0, u(1) = 0$.
We consider the initial value problem

$$ay'' + by' + cy = d \tag{7.2}$$

with $y(0) = z_1$ and $y'(0) = z_2$
The first boundary condition on u forces $z_1 = 0$. However z_2 is arbitrary at this stage.
Because of the linearity of (**??**) it follows that there are constants p and q so that

$$y(1) = p + qz_2$$

[Exercise: prove this]
The values of p and q can be found easily by using an initial value solver applied to (**??**). For example, if we set $z_2 = 0$ then $p = y(1)$ and if $z_2 = 1$ then $q = y(1) - p$. The value of z_2 for which $y(1) = 0$ is then given by

$$z_2 = -p/q.$$

Example 2

Suppose now that we want to solve the nonlinear boundary value problem

$$u'' + u^2 = 1, \quad u(0) = u(1) = 0.$$

as before, we solve the initial value problem

$$y'' + y^2 = 1, \quad y(0) = z_1, \quad y'(0) = z_2, \tag{7.3}$$

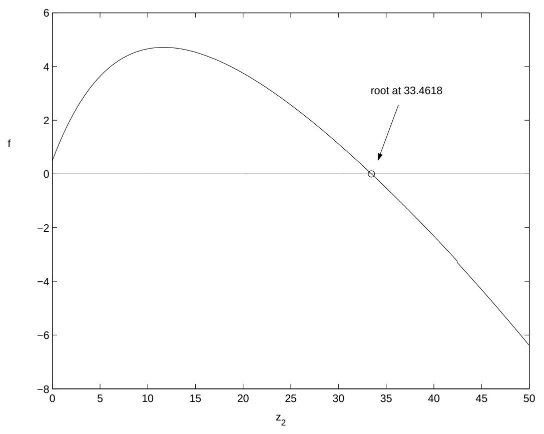
with $z_1 = 0$ forced. In this case we have

$$y(1) = F(z_2)$$

where F is a nonlinear function of z_2. Applying the second boundary condition, $y(1) = 0$, it follows that z_2 must satisfy the equation

$$F(z_2) = 0.$$

The function $F(z_2)$ can be computed by using `ode15s`. This is illustrated below. It is clear that this function has a single root at about $z_2 = 33.46$



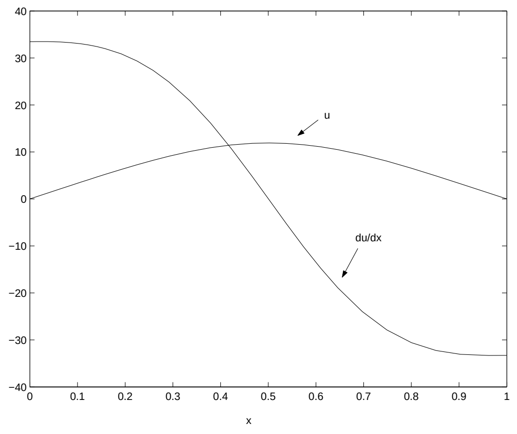
The code to generate the function $F(z_2)$ is given below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                       %
%   Shooting code                       %
%                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function F=funn(x)

[t,u] = ode15s('u2',[0 1],[0 x]);

s  = size(t);
ss = s(1);

F = u(ss,1);

-------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                    %
%  u'' + u^2 = 1                     %
%                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function f=u2(t,u)

f=zeros(2,1);

 f(1) = u(2);
 f(2) = 1-u(1)^2;

-------------------------------------------------------
```

One way to find z_2 is to use the MATLAB command `fzero` given an initial guess of $z_2 = 30$. Notice the way that this code uses the function `funn` computerd using `ode15s`.

```
>>  z2 = fzero('funn',30)

z2  =
33.4618
```

The resulting solution u and u' is illustrated below:



An alternative method, which exploits the structure of the problem, is to use a Newton-Raphson method. As $F(z_2) = y(1)$ it follows that

$$\frac{dF}{dz_2} = \frac{dy(1)}{dz_2} = \phi(z_2)$$

where the function $\phi(x)$ satisfies the *variational equation*

$$\phi'' + 2y\phi = 0, \quad \phi(0) = 0, \quad \phi'(0) = 1.$$

To use this method we take a guess $z_2^{(n)}$ for z_2 and then simultaneously solve the two ODES

$$y'' + y^2 = 1, \quad y(0) = 0, \quad y'(0) = z_2^{(n)}, \quad \phi'' + 2y\phi = 0, \quad \phi(0) = 0, \quad \phi'(0) = 1,$$

using `ode15s`. We then use a Newton-Raphson iteration to improve the guess via the equation

$$z_2^{(n+1)} = z_2^{(n)} - \frac{y(1)}{\phi(1)}.$$

This procedure converges rapidly given a reasonable starting guess.

7.3 Finite Difference Methods

7.3.1 Discretising the second derivative.

Finite difference methods aim to to find the solution at *all interior points* within the interval $[a, b]$. Suppose that we divide up the interval $[a, b]$ into N equal divisions, so that if $h = (b - a)/(N - 1)$ we set $x_i = a + (i - 1)h \quad i = 1, \ldots, N$.
Now we introduce a vector U_i so that

$$U_i \approx u(x_i) \tag{7.4}$$

Now, by using a Taylor series expansion, it follows that

$$\begin{aligned} u(x_{i+1}) &= u(x_i) + hu'(x_i) + \tfrac{h^2}{2}u''(x_i) + \ldots \\ u(x_{i-1}) &= u(x_i) - hu'(x_i) + \tfrac{h^2}{2}u''(x_i) + \ldots \end{aligned}$$

By making a slightly more detailed calculation it follows that

$$\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2} = u''(x_i) + \frac{h^2}{12}u^{iv}(x_i) + \ldots \tag{7.5}$$

We now can use (??) to approximate u'' by

$$u'' \approx \frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} \equiv \frac{\delta^2}{h^2} U_i \tag{7.6}$$

where δ^2 is the iterated central difference operator. The leading error made in (??) when approximating u'' by the iterated central difference is then proportional to $h^2 u^{iv}$. Now suppose that A is the tri-diagonal matrix given by

$$A = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1/h^2 & -2/h^2 & 1/h^2 & \\ \ddots & \ddots & \ddots & \\ & 1/h^2 & -2/h^2 & 1/h^2 \\ 0 & & & 1 \end{bmatrix},$$

If the vector U then satisfies the linear system

$$AU = \begin{bmatrix} \alpha \\ f_2 \\ \vdots \\ f_{N-1} \\ \beta \end{bmatrix} \quad \text{where} \quad f_i \equiv f(x_i) \tag{7.7}$$

then (??) is a discretisation to the second order BVP with Dirichlet boundary conditions which is given by

$$u'' = f(x), \quad u(a) = \alpha, \quad u(b) = \beta.$$

If we replace U by u in (??) the expression (??) implies that we would make an error of order h^2 as $h \to 0$. Note that the structure of A forces $U_1 = \alpha$ and $U_N = \beta$. The vector U can then be found via the operation

$$U = A^{-1} \begin{bmatrix} \alpha \\ f_2 \\ f_{N-1} \\ \beta \end{bmatrix}$$

Provided that the problem is not too irregular it can then be shown that in the limit of $N \to \infty$ $(h \to 0)$ we have

$$U_i = u(x_i) + O(h^2) \qquad \text{as} \quad h \to 0.$$

The solution of the BVP thus becomes (under discretisation) the problem of solving the linear system (??).
If $f = [\alpha, f_2, \dots, f_{N-1}, \beta]'$ then this can be done in MATLAB via the command $A \backslash f$.

Whilst this is simple to use it is not especially efficient as the backslash operator does *not* exploit the special tri-diagonal structure of Matrix A. A *much* more efficient algorithm of complexity $O(N)$ is the Thomas algorithm which exploits this structure and is based on the LU decomposition. This algorithm is described in Assignment 4. (Note, use of the MATLAB sparse matrix routines will speed things up here.)

Efficiency is important as the relatively large error of $O(N^{-2})$ in this method means that we have to take a large value of N to get a reasonable error estimate. More careful discretisations lead to smaller errors at the expense of more work.

7.3.2 Different boundary conditions

We can extend this idea to BVPs with *Neumann* boundary conditions. For example the BVP $u'' = f$ with the boundary conditions

$$u'(a) = \alpha, \; u(b) = \beta.$$

Problems such as this arise frequently in models of heat conduction and electrical flow. For example, the boundary condition $u'(a) = 0$ corresponds to a thermal or electrical insulator. There are many ways to deal with such a derivative boundary condition. In the simplest we approximate

$$u'(a) \quad \text{by} \quad (U_2 - U_1)/h \tag{7.8}$$

This will change the matrix A to the tridiagonal matrix

$$A \equiv \begin{bmatrix} -1/h & 1/h & 0 & \dots 0 \\ 1/h^2 & -2/h^2 & 1/h^2 & \\ & \ddots & \ddots & \ddots \\ & 1/h^2 & -2/h^2 & -1/h^2 \\ 0 & & 0 & 1 \end{bmatrix}$$

and we must consider the solution to the linear system:

$$AU = \begin{bmatrix} \alpha \\ f_2 \\ \vdots \\ f_{N-1} \\ \beta \end{bmatrix} \tag{7.9}$$

The linear equation (**??**) is then a discretisation of the BVP

$$u'' = f \qquad u'(a) = \alpha, \qquad u(b) = \beta.$$

The expression (**??**) is only accurate to $O(h)$ as an approximation to $u'(a)$. We can improve this by using the approximation

$$u'(a) \approx (4U_2 - U_3 - 3U_1)/2h$$

Exercise. Show that this expression is accurate to $O(h^2)$

This leads to a slightly different matrix A. Unfortunately the resulting matrix is no longer tri-diagonal and we cannot use the Thomas algorithm to invert it.

A special case of Neumann boundary conditions arises when $u'(a) = 0$. In this case we can introduce a "ghost" point U_0 approximating $u(a - h)$. The boundary condition implies that to $0(h^3)$ we have $U_0 = U_2$. At the point $x = 0$ we have

$$u''(a) \approx \frac{U_0 + U_2 - 2U_1}{h^2} = \frac{2U_2 - 2U_1}{h^2} \tag{7.10}$$

This approximation to $u''(a)$ is correct to $O(h^2)$. (Can you show this?)
The resulting matrix A is then given by

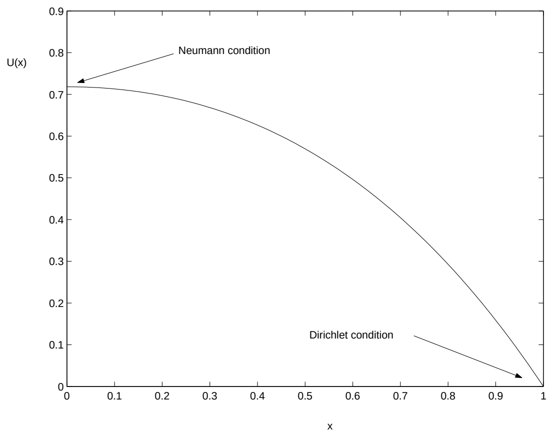
$$A = \frac{1}{h^2} \begin{bmatrix} -2 & 2 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & \ddots \end{bmatrix}$$

Note that in this case A is tri-diagonal, and we can again solve the linear system by using the Thomas algorithm.

%%

```
%                                                            %
%   Code to solve the Neumann two-point boundary value      %
%   problem:                                                 %
%                                                            %
%   u'' = -exp(x),  u'(0) = 0, u(1) = 0.                     %
%                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%
%  Set up the mesh
%

N = 101;

h = 1/(N-1);

x = [0:h:1];

%
%  Set up the matrix
%

A = diag(ones(N-1,1),-1) + diag(ones(N-1,1),1) - 2*diag(ones(N,1));

A = A/h^2;

%
%  Neumann boundary condition at x = 0
%

A(1,2) = 2/h^2;

%
% Dirichlet boundary condition at x = 1
%

A(N,N-1) = 0;
A(N,N)   = 1;


%
%  Set up the right hand side
%

f = -exp(x)';

%
%  Modify this to allow for the Dirichlet boundary condition
%

f(N) = 0;
```

```
%
%  Solve the system (without using the Thomas algorithm)
%

U = A\f;

%
% Plot the solution
%

plot(x,U)
```



A second example of differential equations occurs for BVPs with *periodic boundary conditions* for which $u(a) = u(b)$ and $u'(a) = u'(b)$. Periodic boundary conditions often arise when looking at those BVPs that describe the *travelling wave* solutions of hyperbolic equations. They also arise naturally when solving BVPs on circles or spheres. A natural example of the latter being weather forecasting on the whole globe. Often BVPs with periodic boundary conditions are solved using *spectral methods* in which the solution is expressed as a combination of trigonometric functions. However they can also be solved by using finite difference methods. Exploiting periodicity we have

$$u''(a) \approx \frac{u(a+h) + u(a-h) - 2u(a)}{h^2} = \frac{u(a+h) + u(b-h) - 2u(a)}{h^2} \tag{7.11}$$

which we approximate by

$$u''(a) \approx \frac{U_2 + U_{N-1} - 2U_1}{h^2}$$

Setting $\mathbf{U} = [U_1, \ldots, U_{N-1}]'$ (noting that $U_1 = U_N$) the resulting matrix A is given by

$$A = \begin{bmatrix} -2/h^2 & 1/h^2 & 0\ldots & 0 & 1/h^2 \\ 1/h^2 & \ddots & \ddots & & 0 \\ 0 & & \ddots & \ddots & \\ 1/h^2 & \ldots & \ldots & 1/h^2 & -2/h^2 \end{bmatrix}$$

The resulting discretisation of u'' then has an error of $O(h^2)$ as before. The matrix A is not tri-diagonal, but its special periodic structure means that it can be inverted quickly by using the FFT.

7.3.3 Adding in convective terms

In many applications we meet *convection- diffusion problems* which typically take the form

$$\epsilon u'' + u' = f \quad \text{with } \epsilon \text{ small.}$$

These arise in fluid mechanics problems, where ϵ is a measure of the *viscosity* of the fluid. The most accurate discretisation of u' is given by the *central difference* discretisation

$$u' \approx \frac{\delta U}{2h} \equiv \frac{U_{i+1} - U_{i-1}}{2h}$$

A simple calculation shows that the central difference gives an error $O(h^2)$ when approximating u'. However this discretisation of u' leads to problems. In particular A^{-1} may be nearly singular and ill-conditioned. If ϵ is zero then A has an eigenvector of the form $e = (1, -1, 1, -1, 1, -1, \ldots)$ with zero eigenvalue eigenvalue. Check thisso that A^{-1} is singular.
For small ϵ, A continues to have a similar eigenvector to ϵ with a small eigenvalue. When inverting A the contribution of e to the solution is greatly amplified, and can lead to an oscillatory contribution to the solution. This instability is *BAD* but can easily be detected as it is on a grid scale. i.e all oscillations in the solution are the same size as the mesh.
Oscillations can be avoided by taking h small enough to avoid this instability so that

$$\boxed{h \leq 2\epsilon}$$

However, this is a very restrictive condition if ϵ is small.

One way to avoid the instability when $\epsilon > 0$ is to replace u' by its *upwind difference*

$$\frac{U_{i+1} - U_i}{h}$$

This is a less accurate approximation of u' than the central differenceas

$$\frac{U_{i+1} - U_i}{h^2} = u' + O(h)$$

However the resulting matrix A is *much* better conditioned and inverting it does *not* lead to spurious oscillations in the solution What this approximation is doing is to exploit a natural flow of information in the system.

This is not always a good solution as often we can't tell in advance if we need to use the upwind difference or the *downwind difference* $\frac{U_i - U_{i-1}}{h}$. However, like a stiff solver, the use of the upwind difference allows us to use a step size governed by *accuracy* rather than *stability.* An example of the effect of using these different discretisations is given in Assignment 4.

7.4 Nonlinear Problems.

So far we have looked at linear two point BVPs. However,

$$\boxed{\text{MOST PROBLEMS ARE NONLINEAR}}$$

An important class of nonlinear two point BVPs (called semilinear problems) take the form

$$a(x)u'' + b(x)u' + f(x,u) = 0, \tag{7.12}$$

and we need to develop techniques to solve them. Two examples of such problems are given by the differential equation

$$u'' + e^{u/(1+u)} = 0,\ u(a) = u(b) = 0$$

which models combustion in a chemically reacting material, and

$$u'' + \frac{2}{x}u' + k(x)u^5 = 0,\ u'(0) = 0, \quad u(\infty) = 0,\ u > 0,$$

which describes the curvature of space by a spherical body.

A special case is given by the equation

$$u'' + f(u) = 0, \quad u(a) = \alpha, \quad u(b) = \beta. \tag{7.13}$$

Discretising the differential equation (**??**) leads to a set of nonlinear equations of the form

$$AU + f(U) = 0 \tag{7.14}$$

where A is the matrix given earlier and the vector f is given by

$$f_i \equiv f(U_i).$$

As with most nonlinear problems, we solve this system by *iteration* starting with an initial guess although, be warned, the system may have one, none or many solutions. Perhaps the most effective such method is the Newton-Raphson algorithm. Suppose that $U^{(n)}$ is an approximate solution of (**??**), we define the residual $R^{(n)}$ by

$$AU^{(n)} + f(U^{(n)}) = R^{(n)}. \tag{7.15}$$

The size of R^n is a measure of the quality of this solution.

Now if we define the Jacobian of the nonlinear function f by

$$J_{ij} = \partial f_i / \partial U_j$$

the "linearisation" L of (**??**) acting on a vector φ is given by

$$L\varphi \equiv A\varphi + J\varphi.$$

The Newton-Raphson iteration updates $U^{(n)}$ to $U^{(n+1)}$ via the iteration

$$U^{(n+1)} = U^{(n)} - L^{-1}R^{(n)}$$

Here the vector $W \equiv L^{-1}R^{(n)}$ can be found by solving the linear system

$$AW + JW = R^{(n)}.$$

This algorithm converges rapidly (often in about 5 iterations) if the initial guess $U^{(0)}$ is close to the true solution. Finding such an initial guess can be difficult for a general problem and often requires some a-priori knowledge of the solution.

The Fortran code AUTO uses the Newton-Raphson method coupled to a path-following (homotopy) method to find the solution of a version of the nonlinear systems arising from discretisations of nonlinear BVPs obtained by using collocation.

7.5 Other methods

At present *collocation* is the best method for solving 2pt BVPs. This is a bit like the Finite difference method but uses much higher order polynominals. It is also closely related to implicit Runge-Kutta methods. However, the collation method has no easy extension to higher dimensions. In higher dimensions two-point BVPs generalise to elliptic partial differential equations. For such problems the finite element method is the *mainly* used solution procedure, although spectral methods are widely used for problems with a relatively simple geometry e.g. weather forecasting on the sphere.

CJB 2006