Chap 4: Preliminary material
Central difference approximations of u' and u''

Let $u_i = u(x_i)$ and consider Taylor expansions about x_i of $u_{i-1} = u(x_{i-1}) = u(x_i - h)$ and $u_{i+1} = u(x_i + h) = u(x_i + h)$.

$$\begin{aligned} u_{i+1} &= u_i + hu_i' + \frac{h^2}{2}u_i'' + \frac{h^3}{6}u_i''' + \frac{h^4}{24}u_i'''' + \cdots \\ u_{i-1} &= u_i - hu_i' + \frac{h^2}{2}u_i'' - \frac{h^3}{6}u_i''' + \frac{h^4}{24}u_i'''' + \cdots . \end{aligned}$$

Adding and subtracting gives

$$\begin{aligned} u_{i+1} + u_{i-1} &= 2\left(u_i + \frac{h^2}{2}u_i'' + \frac{h^4}{24}u_i'''' + \cdots\right) \\ u_{i+1} - u_{i-1} &= 2\left(hu_i' + \frac{h^3}{6}u_i''' + \cdots\right). \end{aligned}$$

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u_i'' + \mathcal{O}(h^2), \quad \frac{u_{i+1} - u_{i-1}}{2h} = u_i' + \mathcal{O}(h^2).$$

Chap 4: The two-point BVP

$$u''(x) = p(x)u'(x) + q(x)u(x) + r(x), \quad a < x < b,$$
$$u(a) = g_1, \quad u(b) = g_2.$$

The FD approximation – a summary

With a uniform mesh with $h = (b-a)/N$, $x_i = a + ih$, $i = 0, 1, \ldots, N$ and $U_i \approx u(x_i)$ the central difference finite difference approximation involves the following.

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} = p_i \left(\frac{U_{i+1} - U_{i-1}}{2h} \right) + q_i U_i + r_i,$$
$$i = 1, 2, \ldots, N-1.$$

$$U_0 = g_1 \quad \text{and} \quad U_N = g_2.$$

The "continuous" problem for $u(x)$, $a \le x \le b$ is approximated by a "discrete" problem involving $U_0, U_1, \ldots, U_N$.

The local truncation error

The local truncation error is concerned with how nearly the exact solution satisfies the difference equations that determine the finite difference approximation. It is defined as follows for $i = 1, \ldots, N-1$.

$$L_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - \left(p_i \left(\frac{u_{i+1} - u_{i-1}}{2h} \right) + q_i u_i + r_i \right) = \mathcal{O}(h^2).$$

The linear system $A\underline{U} = \underline{c}$

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & \ddots & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & a_{N-2,N-1} \\ 0 & \cdots & 0 & a_{N-1,N-2} & a_{N-1,N-1} \end{pmatrix},$$

$$a_{i,i-1} = -1 - \frac{hp_i}{2}, \quad a_{ii} = 2 + h^2 q_i, \quad a_{i,i+1} = -1 + \frac{hp_i}{2}.$$

$$\begin{aligned} c_1 &= -h^2 r_1 + \left(1 + \frac{hp_1}{2}\right) g_1, \\ c_i &= -h^2 r_i, \quad 2 \le i \le N-2, \\ c_{N-1} &= -h^2 r_{N-1} + \left(1 - \frac{hp_{N-1}}{2}\right) g_2. \end{aligned}$$

It is $\mathcal{O}(N)$ storage and it $\mathcal{O}(N)$ operations to solve for $\underline{U}$.

The system in the special case $u'' = r$

When $p(x) = q(x) = 0$ we have $U_0 = g_1$, $U_N = g_2$ and

$$\frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} = r_i, \quad i = 1, 2, \ldots, N-1.$$

The tri-diagonal system is as follows.

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_{N-2} \\ U_{N-1} \end{pmatrix} = \begin{pmatrix} -h^2 r_1 + g_1 \\ -h^2 r_2 \\ \vdots \\ -h^2 r_{N-2} \\ -h^2 r_{N-1} + g_2 \end{pmatrix}.$$

It is $\mathcal{O}(N)$ storage and it $\mathcal{O}(N)$ operations to solve for $\underline{U}$.

Matlab, the `spdiags` function and \

For the problem $u'' = r$, and assuming that a, b, N, g_1 and g_2 have been given and $r(x)$ is available, the statements to get $\underline{U}$ can be as follows.

```
% Uniform mesh
x=linspace(a, b, N+1); x=x(:);
h=(b-a)/N;

% set-up the tri-diagonal matrix A
N1=N-1;
o=ones(N1, 1);
A=spdiags([-o, 2*o, -o], -1:1, N1, N1);

% set-up the rhs vector c
c=-h*h*r(x(2:N));
c(1)=c(1)+g1;
c(N1)=c(N1)+g2;

% solve the equations
U=[g1; zeros(N1, 1); g2];
U(2:N)=A\c;
```

Creating a version as a function

By starting with a statement which contains `function` we get a function file version.

```
function[x, U]=fd_solver(a, b, N, r, g1, g2)

x=linspace(a, b, N+1); x=x(:);
h=(b-a)/N;

% ..statements as before but not shown here

% solve the equations
U=[g1; zeros(N1, 1); g2];
U(2:N)=A\c;
```

It is good practice to add suitable comments and in particular to explain the input and output arguments.

Illustrating tri-diagonal matrices in Matlab

```
% small tri-diagonal matrix
n=4;
o=ones(n, 1);
A=spdiags([-o 2*o -o], -1:1, n, n);
[L, U]=lu(A);

FA=full(A)
FL=full(L)
FU=full(U)
```

The LU factorization of A when $n = 4$

```
FA =
     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
FL =
    1.0000         0         0         0
   -0.5000    1.0000         0         0
         0   -0.6667    1.0000         0
         0         0   -0.7500    1.0000
FU =
    2.0000   -1.0000         0         0
         0    1.5000   -1.0000         0
         0         0    1.3333   -1.0000
         0         0         0    1.2500
```

Testing the solver in Matlab

We create a problem for which we know the solution $u(x)$.

$$[a, b] = [0, \pi], \quad u(x) = \cos(3x), \quad r(x) = u''(x) = -9\cos(3x).$$

Here $r(x)$ is the rhs function to use in the numerical scheme.

```
% Test problem
a=0; b=pi;
uex =@(x) cos(3*x);
r   =@(x) -9*cos(3*x);
g1=uex(a);
g2=uex(b);
N=4;

[x, U]=fd_solver(a, b, N, r, g1, g2)
```

Trying the method for $N = 2^k$, $k = 2, 3, \ldots, 16$

```
% Test problem set-up statements
a=0; b=pi;
uex =@(x) cos(3*x);
r   =@(x) -9*cos(3*x);
g1=uex(a);
g2=uex(b);

for k=2:16
  N=2^k;
  [x, U]=fd_solver(a, b, N, r, g1, g2);
  fprintf('N=%5d, error=%12.4e\n', ...
            N, norm(uex(x)-U, inf));
end
```

The output from the previous script

The 15 lines of output are as follows.

```
N=    4, error=  7.5570e-01
N=    8, error=  1.4986e-01
N=   16, error=  3.9892e-02
N=   32, error=  9.8432e-03
N=   64, error=  2.4528e-03
N=  128, error=  6.1270e-04
N=  256, error=  1.5314e-04
N=  512, error=  3.8288e-05
N= 1024, error=  9.5720e-06
N= 2048, error=  2.3930e-06
N= 4096, error=  5.9825e-07
N= 8192, error=  1.4957e-07
N=16384, error=  3.7401e-08
N=32768, error=  9.4565e-09
N=65536, error=  2.3234e-09
```

Note that as `N` is doubled the error decreases by about 4 illustrating that the error decreases like h^2.

The ratios and a neater table version

To save the errors, compute the ratios and have a neat table you can do the following.

```
% ..set-up as before
fprintf('%5s  %12s  %10s\n', ...
        'N', 'Error', 'Ratios');
e=zeros(1, 16);
for k=2:16
  N=2^k;
  [x, U]=fd_solver(a, b, N, r, g1, g2);
  e(k)=norm(uex(x)-U, inf);
  if k==2
    fprintf('%5d  %12.4e\n', N, e(k));
  else
    fprintf('%5d  %12.4e  %10.6f\n',...
             N, e(k), e(k-1)/e(k));
  end
end
```

The neater table output

The previous program creates the following table.

N	Error	Ratios
4	7.5570e-01	
8	1.4986e-01	5.042612
16	3.9892e-02	3.756714
32	9.8432e-03	4.052725
64	2.4528e-03	4.013054
128	6.1270e-04	4.003256
256	1.5314e-04	4.000813
512	3.8288e-05	3.999763
1024	9.5720e-06	4.000051
2048	2.3930e-06	4.000012
4096	5.9825e-07	3.999998
8192	1.4957e-07	3.999873
16384	3.7401e-08	3.998959
32768	9.4565e-09	3.955107
65536	2.3234e-09	4.070037

It is the matching of the widths in the `fprintf` statements which gives the alignment.

Chapter 4 summary

- Most differential equations do not have a "closed form" solution but you can still approximate the solution using numerical methods.
- To understand the finite difference approximations to derivatives you need to understand Taylor series expansions, e.g.

$$u_{i+1} = u(x_i + h) = u_i + hu_i' + \frac{h^2}{2!}u_i'' + \frac{h^3}{3!}u_i''' + \frac{h^4}{4!}u_i'''' + \cdots$$

$$u_{i-1} = u(x_i - h) = u_i - hu_i' + \frac{h^2}{2!}u_i'' - \frac{h^3}{3!}u_i''' + \frac{h^4}{4!}u_i'''' + \cdots$$

Re-arranging gives the following.

$$u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - \frac{h^2}{12}u''''(x_i) + \mathcal{O}(h^4).$$

$$u_i' = \frac{u_{i+1} - u_{i-1}}{2h} - \frac{h^2}{6}u'''(x_i) + \mathcal{O}(h^4).$$

The central difference approximations follow from this.

Chapter 4 summary continued

- Derivative boundary conditions can also be handled, i.e. we can consider

$$u''(x) = p(x)u'(x) + q(x)u(x) + r(x), \quad a < x < b,$$
$$u(a) = g_1, \quad u'(b) = g_3.$$

  By appropriately combining Taylor expansions we get

$$4u(b-h) - u(b-2h) = 3u(b) - 2hu'(b) + \frac{4h^3}{6}u'''(b) + \cdots$$

  See the exercise sheet for other finite difference schemes with some involving more than 3 points.

- The matrix of the linear system to solve is tri-diagonal. When the interval $[a, b]$ is divided into N equal sub-intervals of width $h = (b-a)/N$ the storage required grows like $\mathcal{O}(N)$ and the amount of computation also grows like $\mathcal{O}(N)$. An efficient implementation can be done in Matlab.

Comments about the initial value problem

The scalar case is

$$u' = f(t, u(t)), \quad u(t_0) = u_0.$$

Consider Taylor expansions about t_n evaluated at $t_n + h$.

$$\begin{aligned} u_{n+1} &= u_n + hu'_n + \mathcal{O}(h^2), \\ &= u_n + hu'_n + \frac{h^2}{2}u''_n + \mathcal{O}(h^3), \\ &= u_n + hu'_n + \frac{h^2}{2}u''_n + \frac{h^3}{6}u'''_n + \mathcal{O}(h^4), \end{aligned}$$

As $u(t)$ satisfies the ODE we have the following.

$$\begin{aligned} u'(t) &= f(t, u(t)), \\ u''(t) &= \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}u'\right)(t, u(t)), \end{aligned}$$

by the chain rule of partial differentiation. The complexity of the derivatives increases considerable in the general case for each higher derivative.

Taylor' series methods

Let $t_n = t_0 + nh$, $n = 0, 1, \ldots$ with $t_f - t_0 = Nh$. Let $U_n \approx u_n = u(t_n)$.

In all cases $U_0 = u_0$. These methods need f and the partial derivatives (depending on the scheme).

Euler's method

$$U_{n+1} = U_n + hf(t_n, U_n), \quad n = 0, 1, 2, \ldots.$$

Local truncation error is $\mathcal{O}(h^2)$. Accumulated error is $\mathcal{O}(h)$.

The TS(2) method

$$U_{n+1} = U_n + hf(t_n, U_n) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t}(t_n, U_n) + f(t_n, U_n)\frac{\partial f}{\partial u}(t_n, U_n)\right),$$

$$n = 0, 1, 2, \ldots.$$

Local truncation error is $\mathcal{O}(h^3)$. Accumulated error is $\mathcal{O}(h^2)$.

Huen's method for scalars

This has the same accuracy as the TS(2) scheme and only needs function values.

Heun's method for the scalar problem. Start with $U_0 = u(0)$.

For $n = 0, 1, 2, \ldots$

$$k_1 = f(t_n, U_n),$$

$$k_2 = f(t_n + h, U_n + hk_1),$$

$$U_{n+1} = U_n + \frac{h}{2}(k_1 + k_2).$$

End For loop

This can be generalised to the case of systems.

The Runge-Kutta method of order 4 for systems

The best known of the Runge Kutta schemes is the following scheme which has an accumulated error of $\mathcal{O}(h^4)$. In the case of systems the problem is

$$\underline{u}'(t) = \underline{f}(t, \underline{u}(t)), \quad \underline{u}(0) = \underline{u}_0$$

and the scheme is as follows.

For $n = 0, 1, 2, \ldots$

$$\underline{k}_1 = \underline{f}(t_n, \underline{U}_n),$$
$$\underline{k}_2 = \underline{f}(t_n + h/2, \underline{U}_n + (h/2)\underline{k}_1),$$
$$\underline{k}_3 = \underline{f}(t_n + h/2, \underline{U}_n + (h/2)\underline{k}_2),$$
$$\underline{k}_4 = \underline{f}(t_n + h, \underline{U}_n + h\underline{k}_3),$$
$$\underline{U}_{n+1} = \underline{U}_n + \frac{h}{6}(\underline{k}_1 + 2\underline{k}_2 + 2\underline{k}_3 + \underline{k}_4).$$

End For loop

This method is from around 1900 and is still used today. The Matlab solver `ode45()` makes use of the RK4 scheme as part of what it does.