6G6Z3002 Computational Methods in Ordinary Differential Equations

Runge-Kutta Methods

Lecture Notes

Dr Jon Shiach

2020 – 2021

Department of Computing and Mathematics

Contents

Initial Value Problems

Learning outcomes

On successful completion of this chapter readers will be able to:

- Identify an initial value problem;
- Derive the Euler method and apply it to solve first-order ordinary differential equations;
- Express a higher-order ODE as a system of first-order ODEs;
- Apply an ODE solver to solve a system of ODEs.

An Initial Value Problem (IVP) is written as an Ordinary Differential Equation (ODE) where the initial solution at the lower boundary of the domain is known. For example,

$$y' = f(t, y), \qquad t \in [t_{\min}, t_{\max}], \qquad y(t_{\min})$$

Here a first-order ODE $y' = f(t, y)$ is defined over the domain $t \in [t_{\min}, t_{\max}]$ and the initial solution $y(t_{\min})$ is the known value y_0.

In most real-world practical applications, IVPs cannot be solved using analytical methods so instead we use computational methods called **ODE solvers** to approximate the solutions. The simplest ODE solver is the Euler method.

The Euler method

The Euler method for solving the first-order Ordinary Differential Equation (ODE) can be derived very easily by truncating the Taylor series after the first-order term

$$y_{n+1} = y_n + hf(t_n, y_n),$$

where the subscript notation used is $y_{n+1} = y(t + h)$, $y_n = y(t_n)$.

To apply the Euler method to solve an IVP we loop through the domain and calculate the values of y_{n+1} using the Euler method with the known values t_n and y_n.

Example 1

Calculate the first two steps of the Euler method when used to solve the following IVP

$$y' = \sin^2(t)y, \qquad t \in [0, 5], \qquad y(0) = 1.$$

using a step length $h = 0.5$.

Since we have $t_0 = 0$ and $y_0 = 1$ then

$$y_1 = y_0 + h\sin^2(t_0)y_0 = 1 + 0.5\sin^2(0)(1) = 1, \qquad t_1 = t_0 + h = 0 + 0.5 = 0.5,$$
$$y_2 = y_1 + h\sin^2(t_1)y_1 = 1 + 0.5\sin^2(0.5)(1) = 1.1149, \qquad t_2 = t_1 + h = 0.5 + 0.5 = 1.0.$$

Example 2

Calculate the solution of the IVP from example 1 over the whole domain $t \in [0, 5]$ using the Euler method with a step length of $h = 0.5$ and compare it to the exact solution given by

$$y = e^{(t-\sin(t)\cos(t))/2}$$

The function called `euler` below calculates the solution to an IVP using the Euler method. The input arguments are `f` is the name of the function the defines the ODE, `tspan` is a two-element array containing the lower and upper bounds of the t domain, `y0` is the initial value of y at the lower bound and h is the step length.

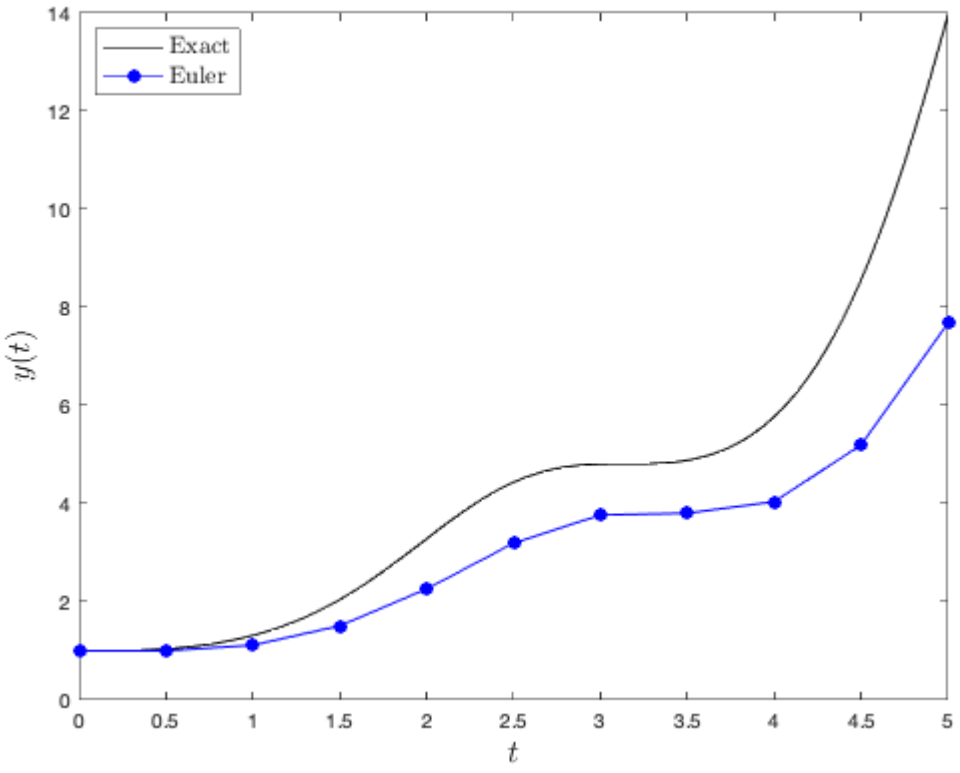
```
function [t, y] = euler(f, tspan, y0, h)

% Calculuates the solution to an IVP using the Euler method

% Initialise output arrays
nsteps = floor((tspan(2) - tspan(1)) / h);
t = zeros(nsteps, 1);
y = zeros(nsteps, 1);
t(1) = tspan(1);
y(1) = y0;

% Solver loop
for n = 1 : nsteps
    y(n + 1) = y(n) + h * f(t(n), y(n));
    t(n + 1) = t(n) + h;
end

end
```

The program below invokes the Euler method to solve this IVP.

```
% Define ODE function
f = @(t, y) sin(t)^2 * y;

% Define exact solution
exact = @(t) exp(0.5 * (t - sin(t) .* cos(t)));

% Define IVP parameters
tspan = [ 0, 5 ];
y0 = 1;
h = 0.5;

% Invoke Euler method to solve IVP
[t, y] = euler(f, tspan, y0, h);

% Output solution table
table = sprintf(' t    Euler   Exact \n--------------------\n');
for n = 1 : length(t)
    table = [table , sprintf('%3.2f %7.4f %7.4f\n', t(n), y(n), exact(t(n)))];
end
fprintf(table)
```

```
   t    Euler   Exact
 ------------------
 0.00  1.0000  1.0000
 0.50  1.0000  1.0404
 1.00  1.1149  1.3135
 1.50  1.5096  2.0436
 2.00  2.2607  3.2845
 2.50  3.1953  4.4359
 3.00  3.7675  4.8059
 3.50  3.8050  4.8830
 4.00  4.0391  5.7699
 4.50  5.1958  8.5589
 5.00  7.6783 13.9573
```

```
% Plot solutions
t1 = linspace(tspan(1), tspan(2), 100);
plot(t1, exact(t1), 'k-')
hold on
plot(t, y, 'bo-', 'markerfacecolor', 'blue')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'Euler');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



Here we can see that the solution using the Euler method deviates from the exact solution. This is because the Euler method is a first-order method and the truncation errors at each step accumulate as we step through the solution. One way to improve our solution is to use a smaller step length.

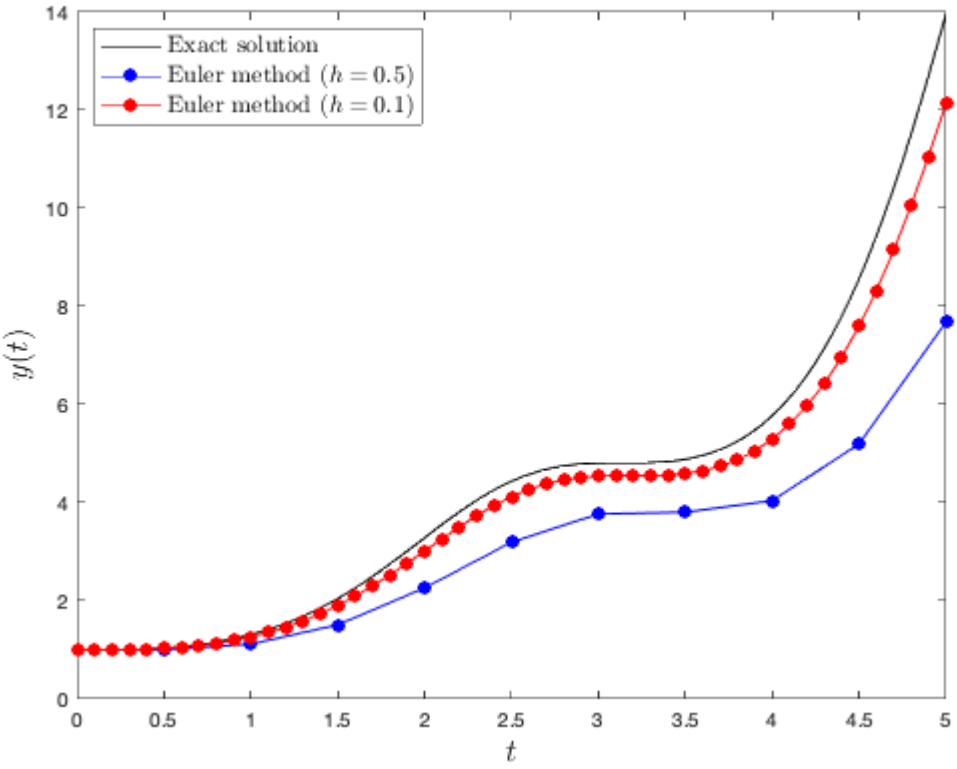
Example 3

Repeat the solution of the IVP in example 2 using a step length of $h = 0.1$ and compare the solution with the exact solution and the one obtained using $h = 0.5$.

```
% Invoke Euler method to solve IVP
[t1, y1] = euler(f, tspan, y0, 0.5);
[t2, y2] = euler(f, tspan, y0, 0.1);

% Output solution table
table = ' t    h=0.5   h=0.1   Exact \n--------------------------\n';
for n = 1 : length(t1)
    table = [table , sprintf('%3.2f %7.4f %7.4f %7.4f\n', ...
        t1(n), y1(n), y2(1 + 5*(n-1)), exact(t1(n)))];
end
fprintf(table)
```

```
  t    h=0.5   h=0.1   Exact
 --------------------------
 0.00  1.0000  1.0000  1.0000
 0.50  1.0000  1.0291  1.0404
 1.00  1.1149  1.2626  1.3135
 1.50  1.5096  1.9018  2.0436
 2.00  2.2607  3.0133  3.2845
 2.50  3.1953  4.1223  4.4359
 3.00  3.7675  4.5394  4.8059
 3.50  3.8050  4.5919  4.8830
 4.00  4.0391  5.2940  5.7699
 4.50  5.1958  7.5957  8.5589
 5.00  7.6783 12.1225 13.9573
```

```
% Plot solutions
t3 = linspace(tspan(1), tspan(2), 100);
plot(t3, exact(t3), 'k-')
hold on
plot(t, y, 'bo-', 'markerfacecolor', 'b')
plot(t2, y2, 'ro-', 'markerfacecolor', 'r')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact solution', 'Euler method ($h=0.5$)', 'Euler method ($h=0.1$)');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



Here we can see that although using a smaller step length has increased the accuracy of the solution it still does not match the exact solution. The Euler method is easy to derive and apply but is not very accurate.

Solving higher-order ordinary differential equations

The numerical methods that are applied to calculate the solutions to ODEs can only be applied to first-order ODEs. To apply them to higher-order ODEs we first need to rewrite them as a system of first-order ODEs.

Consider the Nth-order ODE

$$y^{(N)} = f(t, y, y', y'', \dots, y^{(N-1)}).$$

If we let $y_1 = y$, $y_2 = y'$, $y_3 = y''$ and so on up to $y_N = y^{(N-1)}$ then we have

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= y_3, \\ &\vdots \\ y_N' &= f(t, y_1, y_2, y_3, \dots, y_N). \end{aligned}$$

This is a system of N first-order ODEs. We can apply our numerical methods for solving ODEs to each equation in the system to give an equivalent solution to the $n$nth-order ODE.

Example 4

Rewrite the following third-order ODE as a system of three first-order ODEs

$$y''' + yy'' - 2y' + ty'' - 10 = 0.$$

Let $y_1 = y$, $y_2 = y'$, $y_3 = y''$ then we can rewrite this ODE using

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= y_3, \\ y_3' &= -y_1 y_3 + 2y_2 - t y_1 y_2 + 10. \end{aligned}$$

Solving systems of ordinary differential equations

Consider a system of N first-order ODEs written in the form

$$\begin{aligned} y_1' &= f_1(t, y_1), \\ y_2' &= f_2(t, y_2), \\ &\vdots \\ y_N' &= f_N(t, y_N). \end{aligned}$$

Let $\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}$ and $F(t, \mathbf{y}) = \begin{pmatrix} f_1(t, y_1) \\ f_2(t, y_2) \\ \vdots \\ f_N(t, y_N) \end{pmatrix}$ then we can write the system in vector form

$$\mathbf{y}' = F(t, \mathbf{y}).$$

Example 5

Calculate two steps of the Euler method used to solve the following IVP

$$y'' + y = 0, \qquad t \in [0, 10], \qquad y(0) = 2, \qquad y'(0) = 0,$$

using a step length of $h = 0.1$.

First, we need to rewrite the second-order ODE as two first-order ODEs. Let $y_1 = y$ and $y_2 = y'$ then

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= 2, \\ y_2' &= -y_1, & y_2(0) &= 0, \end{aligned}$$

and we have

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \qquad F(t, \mathbf{y}) = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix}.$$

Calculating the first two steps of the Euler method:

$$\mathbf{y}_1 = \mathbf{y}_0 + hF(t_n, \mathbf{y}_0) = \begin{pmatrix} 2 \\ 0 \end{pmatrix} + 0.1 \begin{pmatrix} 0 \\ -2 \end{pmatrix} = \begin{pmatrix} 2 \\ -0.2 \end{pmatrix},$$
$$\mathbf{y}_2 = \mathbf{y}_1 + hF(t_n, \mathbf{y}_1) = \begin{pmatrix} 2 \\ -0.2 \end{pmatrix} + 0.1 \begin{pmatrix} -0.2 \\ -2 \end{pmatrix} = \begin{pmatrix} 1.98 \\ -0.4 \end{pmatrix}.$$

Example 6

Calculate the solution of the IVP from example 5 over the whole domain $t \in [0, 10]$ using the Euler method with a step length of $h = 0.1$ and compare it to the eact solution given by

$$y = 2\cos(t).$$

The function called `euler2` below is defined at the bottom of this page solves an IVP defined using a system of ODEs. This is very similar to the `euler` function shown earlier with the exception that the y array has N columns instead of just 1.

```
function [t, y] = euler2(f, tspan, y0, h)

% Calculuates the solution to an IVP expressed using a system of first-order
% ODEs using the Euler method

% Initialise output arrays
nsteps = floor((tspan(2) - tspan(1)) / h);
t = zeros(nsteps, 1);
y = zeros(nsteps, 2);
t(1) = tspan(1);
y(1, :) = y0';

% Solver loop
for n = 1 : nsteps
    y(n + 1, :) = y(n, :) + h * f(t(n), y(n, :));
    t(n + 1) = t(n) + h;
end

end
```

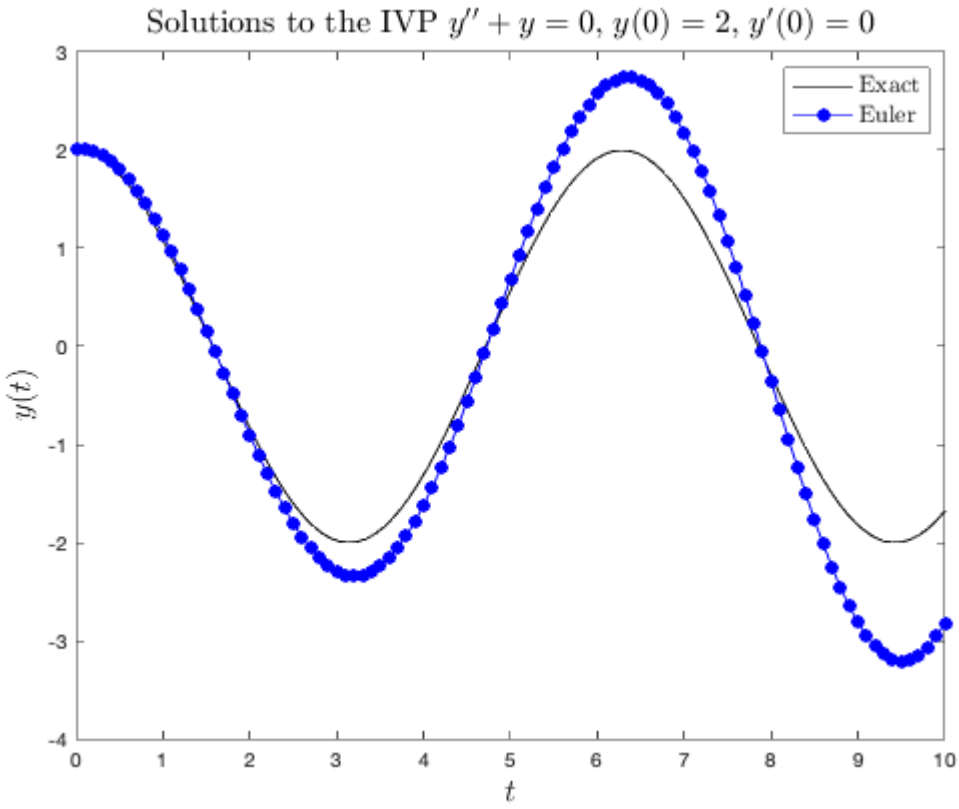
The program below invokes the Euler method to solve this IVP.

```
clear

% Define ODE function
f = @(t, y) [ y(2) , -y(1) ];

% Define exact solution
exact = @(t, y) 2 * cos(t);

% Define IVP parameters
tspan = [ 0, 10 ];
y0 = [ 2 ; 0 ];
h = 0.1;

% Solve IVP using the Euler method
[t, y] = euler2(f, tspan, y0, h);
```

```
% Output solution table
table = ['  t     Euler    Exact\n----------------------\n'];
for n = 1 : 5 : length(t)
    table = [table , sprintf('%5.2f %8.4f %8.4f\n', t(n), y(n,1), exact(t(n)))];
end
fprintf(table)
```

```
   t     Euler    Exact
 ----------------------
  0.00   2.0000   2.0000
  0.50   1.8010   1.7552
  1.00   1.1416   1.0806
  1.50   0.1631   0.1415
  2.00  -0.9060  -0.8323
  2.50  -1.8032  -1.6023
  3.00  -2.2953  -1.9800
  3.50  -2.2387  -1.8729
  4.00  -1.6195  -1.3073
  4.50  -0.5638  -0.4216
  5.00   0.6867   0.5673
  5.50   1.8293   1.4173
  6.00   2.5728   1.9203
  6.50   2.7111   1.9532
  7.00   2.1786   1.5078
  7.50   1.0742   0.6933
  8.00  -0.3550  -0.2910
  8.50  -1.7684  -1.2040
  9.00  -2.8118  -1.8223
  9.50  -3.2054  -1.9943
 10.00  -2.8177  -1.6781
```

```
% Plot solution
t1 = linspace(tspan(2), tspan(1), 100);
plot(t1, exact(t1), 'k-')
hold on
plot(t, y(:, 1), 'bo-', 'MarkerFaceColor', 'b')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
title("Solutions to the IVP $y'' + y = 0$, $y(0)=2$, $y'(0)=0$", ...
    'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'Euler');
set(leg, 'Location', 'Northeast', 'FontSize', 12, 'Interpreter', 'latex')
```



Here we can see that the Euler method solution is deviating away from the exact solution. Once again this is a result of the lower order accuracy of the Euler method.

Summary

- An initial value problem is expressed as an ODE where the solution at the lower bound of the domain is known.
- The Euler method is derived by truncating the Taylor series after the first-order term. It advances the solution over a small step of length h using known values of t_n and y_n to calculate the solution at $y_{n+1} = y(t_n + h)$.
- The Euler method is only first-order accurate, so the solutions tend to be inaccurate unless using a very small value of h.
- Higher-order ODEs can be rewritten as systems of first-order ODEs which can be solved using numerical solvers.

Exercises

1. Using pen and a calculator (i.e., not MATLAB) solve the following IVP using the Euler method with a step length of $h = 0.4$. Write down your solutions correct to 4 decimal places.

$$y' = ty, \qquad t \in [0, 2], \qquad y(0) = 1.$$

2. Reproduce your solutions to question 1 using a MATLAB program.

3. The exam solution to the IVP in question 1 is $y = e^{t^2/2}$. Produce a plot of the Euler solution from question 2 and the exact solution on the same set of axes.

Runge-Kutta Methods

Learning outcomes

On successful completion of this chapter readers will be able to:

- Identify a Runge-Kutta method and express it using a Butcher tableau;
- Distinguish between explicit and implicit Runge-Kutta methods;
- Derive an explicit Runge-Kutta method using the order conditions;
- Apply explicit Runge-Kutta methods to solve an initial value problem.

Runge-Kutta methods are a family of numerical methods for solving a first-order Ordinary Differential Equation (ODE). The general form of a Runge-Kutta method for solving the ODE $y' = f(t, y)$ is

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i, \qquad (1)$$

$$k_i = f(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j). \qquad (2)$$

where t_n is some value of the independent variable t, $y_n = y(t_n)$ is the value of the function y when $t = t_n$, $h = t_{n+1} - t_n$ is the **step length** between two successive values of t, k_i are intermediate **stage values** and s is the number of stages of the method. Runge-Methods are called **single step methods** because they update the solution for the next step y_{n+1} using information from the single step y_n. The other type of method for solving ODEs are **multistep methods** that calculate y_{n+1} using information from multiple steps $y_n, y_{n-1}, y_{n-2}, \ldots$

Butcher tableau

A command method used to express a Runge-Kutta method is to use a **Butcher tableau.** Expanding the summations of the general form of a Runge-Kutta method

$$\begin{aligned}
y_{n+1} &= y + h(b_1 k_1 + b_2 k_2 + \cdots + b_s k_s), \\
k_1 &= f(t_1 + c_1 h, y_n + h(a_{11} k_1 + a_{12} k_2 + \cdots + a_{1s} k_s), \\
k_2 &= f(t_2 + c_2 h, y_n + h(a_{21} k_1 + a_{22} k_2 + \cdots + a_{2s} k_s), \\
&\vdots \\
k_s &= f(t_s + c_s h, y_n + h(a_{s1} k_1 + a_{s2} k_s + \cdots + a_{ss} k_s).
\end{aligned}$$

It is the values of the coefficients a_{ij}, b_i and c_i the define a particular Runge-Kutta method and these values are arranged in tabular form as follows

$$\begin{array}{ccccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ & b_1 & b_2 & \cdots & b_s \end{array}$$

Note that Butcher tableaus are usually represented with partition lines separating the a_{ij}, b_i and c_i values (see examples). MATLAB's LaTeX renderer does not permit this for some reason.

Explicit and implicit Runge-Kutta methods

The stage values of a general Runge-Kutta method are

$$\begin{aligned} k_1 &= f(t_n + c_1 h, y_n + h(a_{11}k_1 + a_{12}k_2 + \cdots + a_{1s}k_s), \\ k_2 &= f(t_n + c_2 h, y_n + h(a_{21}k_1 + a_{22}k_2 + \cdots + a_{2s}k_s), \\ &\vdots \\ k_s &= f(t_n + c_s h, y_n + h(a_{s1}k_1 + a_{s2}k_s + \cdots + a_{ss}k_s). \end{aligned}$$

Here the equation for calculating the value of k_1 includes k_1 on the right-hand side and similar for $k_2, k_3, \ldots, k_s$. These are examples of **implicit** equations and Runge-Kutta methods where the stage values are expressed using implicit equations are known as **Implicit Runge-Kutta** (IRK) methods. To calculate the solution of the stage values of an IRK method involves solving a system of equations which is computationally expensive.

If the summation in equation (2) is altered so the upper limit to the sum is $i-1$, i.e.,

$$k_i = f(t_n + c_1 h, y_n + h\sum_{j=1}^{i-1} a_{ij}k_j),$$

and let $c_1 = 0$ then we have the following equations for calculating the stage values

$$\begin{aligned} k_1 &= f(t_n, y_n)), \\ k_2 &= f(t_n + c_2 h, y_n + h a_{21}k_1)), \\ k_3 &= f(t_n + c_3 h, y_n + h(a_{31}k_1 + a_{32}k_2)), \\ &\vdots \\ k_s &= f(t_n + c_s h, y_n + h(a_{s1}k_1 + a_{s2}k_s + \cdots + a_{s,s-1}k_{s-1})). \end{aligned}$$

These stages values are **explicit** equations where the subject of the equation does not appear on the right-hand side. Runge-Kutta methods where the stages values are calculated using explicit equations are known as **Explicit Runge Kutta** (ERK) methods. Note that we can calculate the stage values in order, i.e., k_1 can be calculated using t_n and y_n; k_2 can be calculated using t_n, y_n and k_1; k_3 can be calculated using t_n, y_n, k_1 and k_2 and so on. This means that ERK methods as more computationally efficient than IRK methods, however in certain situations ERK methods cannot be used and we must then use IRK methods (see notes on Stability).

Butcher tableau for explicit and implicit Runge-Kutta methods

ERK and IRK methods can be easily distinguished looking at the Butcher tableau. The A matrix in the top right region of a Butcher tableau for an ERK method (left) is lower-triangular whereas for an IRK method (right) the main-diagonal and upper triangular elements are non-zero.

$$\begin{pmatrix} 0 & & & \\ a_{21} & & & \\ a_{31} & a_{32} & & \\ \vdots & \vdots & \ddots & \\ a_{s1} & a_{s2} & \cdots & a_{s,s-1} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1s} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2s} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & a_{s3} & \cdots & a_{ss} \end{pmatrix}$$

Derivation of a second-order explicit Runge-Kutta method

A Runge-Kutta method is derived by comparing the Taylor series of the general Runge-Kutta method and ensuring the coefficients a_{ij}, b_i and c_i match those of the Taylor series expansion of the ODE $y' = f(t, y)$.

the second-order Taylor series expansion of $y(t + h)$ is

$$y_{n+1} = y + hy' + \frac{h^2}{2} y''$$

Since $y' = f(t, y)$ and applying the chain rule to differentiate y' we have

$$y'' = f_t(t, y) + f_y(t, y) f(t, y),$$

then the Taylor series becomes

$$\begin{aligned} y_{n+1} &= y + h(t, y) + \frac{h^2}{2} (f_t(t, y) + f_y(t, y) f(t, y)) \\ &= y + \frac{h}{2} f(t, y) + \frac{h}{2} (f(t, y) + hf_t(t, y) + hf_y(t, y) f(t, y)). \end{aligned} \tag{3}$$

The general form of a second-order Runge-Kutta method is

$$y_{n+1} = y_n + hb_1(f_n, y_n) + hb_2 f(t + c_2 h, y + ha_{21} f(t_n, y_n)), \tag{4}$$

The first-order bivariate Taylor series expansion of $f(t_n + c_2 h, y_n + ha_{21} f(t_n, y_n))$ is

$$f(t_n + c_2 h, y_n + ha_{21} f(t_n, y_n)) = f(t_n, y_n) + hc_2 f_t(t_n, y_n) + ha_{21} f(t_n, y_n) f_y(t_n, y_n)$$

which is substituted into equation (4) to give

$$y_{n+1} = y_n + hb_1 f(t_n, y_n) + hb_2(f(t_n, y_n) + hc_2 f_t(t_n, y_n) + ha_{21} f(t_n, y_n) f_y(t_n, y_n)). \tag{5}$$

Equating (3) and (5) we see that

$$b_1 + b_2 = 1,$$
$$c_2 b_2 = \frac{1}{2},$$
$$a_{21} b_2 = \frac{1}{2}$$

Here we have a system of 3 equations expressed in the 4 unknowns a_{21}, b_1, b_2, c_2. These equations are known as the **order conditions** for a second-order Runge-Kutta method. To derive a second-order ERK method we choose a value for one of these coefficients and solve for the others.

Example 1

Derive an ERK method where $c_2 = 1$.

The code below uses the `solve` command to solve the order conditions for a second-order ERK method.

```
% Initialise symbolic variables
syms b1 b2 c2 a21
c2 = 1;

% Define order conditions
eqn1 = b1 + b2 == 1;
eqn2 = c2 * b2 == 1/2;
eqn3 = a21 *b2 == 1/2;

[a21, b1, b2] = solve(eqn1, eqn2, eqn3)
```

```
a21 =
1
b1 =
1/2
b2 =
1/2
```

So $a_{21} = 1$, $b_1 = \frac{1}{2}$ and $b_2 = \frac{1}{2}$ so this second-order ERK method is

$$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2),$$
$$k_1 = f(t_n, y_n),$$
$$k_2 = f(t_n + h, y_n + hk_1),$$

or expressed using a Butcher tableau

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ & \frac{1}{2} & \frac{1}{2} \end{array}$$

This Runge-Kutta method is known as *the* second-order Runge-Kutta method or RK2 for short.

Use of explicit Runge-Kutta methods to solve initial value problems

Runge-Kutta methods are used to solve Initial Value Problems (IVP) of the form

$$y' = f(t, y), \qquad t \in [t_{\min}, t_{\max}], \qquad y(t_{\min}) = y_0,$$

where $y' = f(t, y)$ is a first-order ODE defined over the domain $t \in [t_{\min}, t_{\max}]$ and the initial solution $y(t_{\min})$ is the known value y_0.

To apply the an ERK method to solve an IVP we calculate the stage values $k_1, k_2, \dots, k_s$ using the known values of t_n, y_n and the step length h substituted into the ODE function $f(t, y)$. Then the solution over one step $t_{n+1} = t_n + h$ and $y_{n+1} = y(t_{n+1})$ is then calculated using $k_1, k_2, \dots, k_s$.

Example 2

Calculate the first two steps of the RK2 method to solve the following IVP

$$y' = \sin^2(t)y, \qquad t \in [0, 5], \qquad y(0) = 1.$$

using a step length of $h = 0.5$.

Step 1: $t_0 = 0$ and $y_0 = 1$

$$\begin{aligned} k_1 &= \sin^2(t_0)y_0 = \sin^2(0)(1) = 0, \\ k_2 &= \sin^2(t_0 + h)(y_0 + hk_1) = \sin^2(0 + 0.5)(1 + 0.5(0)) = 0.2298, \\ y_1 &= y_0 + \frac{h}{2}(k_1 + k_2) = 1 + \frac{0.5}{2}(0 + 0.2298) = 1.0575, \\ t_1 &= t_0 + h = 0 + 0.5 = 0.5. \end{aligned}$$

Step 2: $t_1 = 0.5$ and $y_1 = 1.0575$

$$\begin{aligned} k_1 &= \sin^2(t_1)y_1 = \sin^2(0.5)(1.0575) = 0.2431, \\ k_2 &= \sin^2(t_1 + h)(y_1 + hk_1) = \sin^2(0.5 + 0.5)(1.0575 + 0.5(0.2431)) = 0.8348, \\ y_2 &= y_1 + \frac{h}{2}(k_1 + k_2) = 1.0575 + \frac{0.5}{2}(0.2431 + 0.8349) = 1.3269. \end{aligned}$$

Example 3

Calculate the solution of the IVP from example 1 over the whole domain $t \in [0, 5]$ using the Euler method with a step length of $h = 0.5$ and compare it to the exact solution given by

$$y = e^{(t-\sin(t)\cos(t))/2}.$$

The function called `rk2` below calculates the solution to an IVP using the RK2 method. The input arguments are `f` is the name of the function the defines the ODE, `tspan` is a two-element array containing the lower and upper bounds of the t domain, `y0` is the initial value of y at the lower bound and `h` is the step length.

```
function [t, y] = rk2(f, tspan, y0, h)

% Calculates the solution to an IVP using the second-order Runge-Kutta method

% Initialise output arrays
nsteps = floor((tspan(2) - tspan(1)) / h);
t = zeros(nsteps, 1);
y = zeros(nsteps, 1);
t(1) = tspan(1);
y(1) = y0;

% Solver loop
for n = 1 : nsteps
    k1 = f(t(n), y(n));
    k2 = f(t(n) + h, y(n) + h * f(t(n), y(n)));
    y(n + 1) = y(n) + h / 2 * (k1 + k2);
    t(n + 1) = t(n) + h;
end

end
```

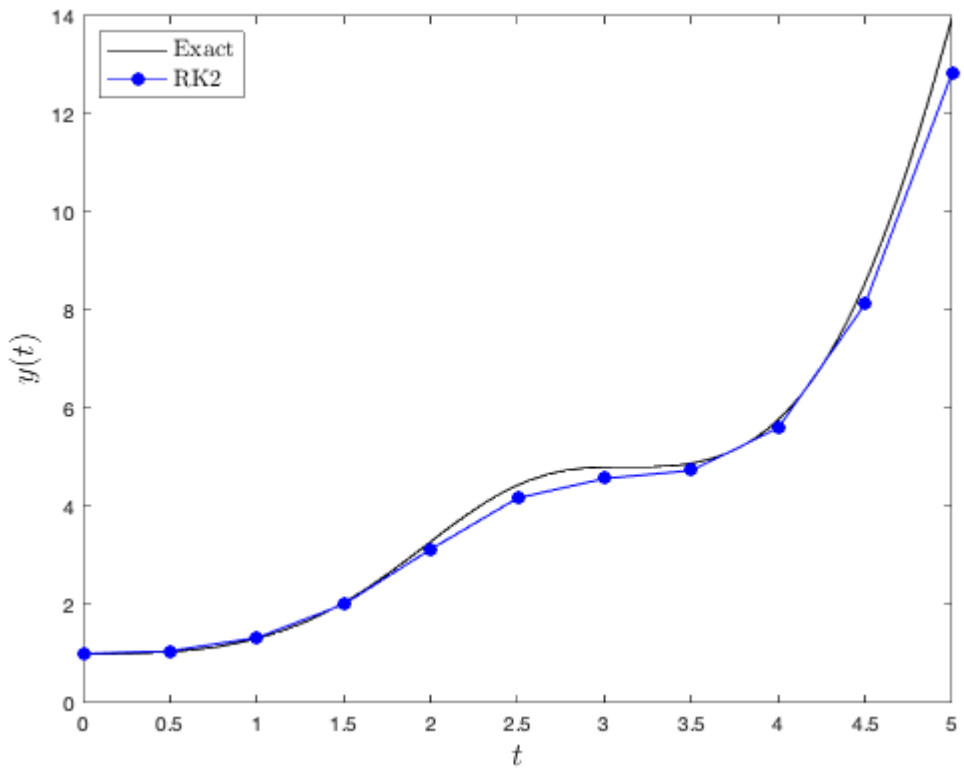
The program below invokes the RK2 method to solve this IVP.

```
clear

% Define ODE function
f = @(t, y) sin(t).^2 .* y;

% Define exact solution
exact = @(t) exp(0.5 * (t - sin(t) .* cos(t)));

% Define IVP parameters
tspan = [ 0, 5 ];
y0 = 1;
h = 0.5;

% Invoke RK2 method to solve IVP
[t, y] = rk2(f, tspan, y0, h);

% Output solution table
table = sprintf(' t     RK2     Exact \n--------------------\n');
for n = 1 : length(t)
    table = [table , sprintf('%3.2f %7.4f %7.4f\n', t(n), y(n), exact(t(n)))];
end
fprintf(table)
```

```
 t     RK2     Exact
--------------------
```

```
0.00  1.0000  1.0000
0.50  1.0575  1.0404
1.00  1.3269  1.3135
1.50  2.0088  2.0436
2.00  3.1302  3.2845
2.50  4.1734  4.4359
3.00  4.5716  4.8059
3.50  4.7364  4.8830
4.00  5.6020  5.7699
4.50  8.1257  8.5589
5.00 12.8273 13.9573
```

```
% Plot solutions
t1 = linspace(tspan(1), tspan(2), 100);
plot(t1, exact(t1), 'k-')
hold on
plot(t, y, 'bo-', 'markerfacecolor', 'blue')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'RK2');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



Here we can see that the RK2 solution follows the approximate shape of the exact solution, however, there are still noticeable errors in the numerical solution. Use of a higher order ERK would improve the accuracy of the numerical results.

Derivation of a fourth-order explicit Runge-Kutta method

The order conditions for a fourth-order ERK method are

$$
\begin{aligned}
b_1 + b_2 + b_3 + b_4 &= 1, \\
b_2 c_2 + b_3 c_3 + b_4 c_4 &= \frac{1}{2}, \\
b_2 c_2^2 + b_3 c_3^2 + b_4 c_4^2 &= \frac{1}{3}, \\
b_2 c_2^3 + b_3 c_3^3 + b_4 c_4^3 &= \frac{1}{4}, \\
b_3 c_3 a_{32} c_2 + b_4 c_4 (a_{42} c_2 + a_{43} c_3) &= \frac{1}{8}, \\
b_3 a_{32} + b_4 a_{42} &= b_2 (1 - c_2), \\
b_4 a_{43} &= b_3 (1 - c_3), \\
0 &= b_4 (1 - c_4).
\end{aligned}
$$

A feature of Runge-Kutta methods is the **row sum condition** that states

$$c_i = \sum_{j=1}^{s} a_{ij},$$

i.e., the value of c_i is equal to the sum of row i of the matrix A. Therefore, the values of a_{21}, a_{31} and a_{41} are

$$
\begin{aligned}
c_2 &= a_{21}, \\
c_3 &= a_{31} + a_{32}, \\
c_4 &= a_{41} + a_{42} + a_{43}.
\end{aligned}
$$

To solve for the unknowns we choose 4 values from b_1, b_2, b_3, b_4, c_2, c_3, c_4 and solve the order conditions for the remaining unknowns.

Example 4

Derive a fourth-order Runge-Kutta method where $c_2 = c_3 = \frac{1}{2}$, $c_4 = 1$ and $b_2 = \frac{1}{3}$.

The code below defines and solves the order conditions and row sum conditions for a fourth-order ERK method.

```
% Initialise symbolic variables
syms a21 a31 a32 a41 a42 a43 b1 b2 b3 b4 c2 c3 c4
c2 = 1/2;
c3 = 1/2;
c4 = 1;
b2 = 1/3;

% Define order conditions
eqn1 = b1 + b2 + b3 + b4 == 1;
eqn2 = b2 * c2 + b3 * c3 + b4 * c4 == 1/2;
eqn3 = b2 * c2^2 + b3 * c3^2 + b4 * c4^2 == 1/3;
eqn4 = b2 * c2^3 + b3 * c3^3 + b4 * c4^3 == 1/4;
eqn5 = b3 * c3 * a32 * c2 + b4 * c4 * (a42 * c2 + a43 * c3) == 1/8;
eqn6 = b3 * a32 + b4 * a42 == b2 * (1 - c2);
eqn7 = b4 * a43 == b3 * (1 - c3);
eqn8 = 0 == b4 * (1 - c4);
```

```
eqn9 = c2 == a21;
eqn10 = c3 == a31 + a32;
eqn11 = c4 == a41 + a42 + a43;

% Solver order conditions
[a21, a31, a32, a41, a42, a43, b1, b3, b4] ...
    = solve(eqn1, eqn2, eqn3, eqn4, eqn5, eqn6, eqn7, eqn8, eqn9, eqn10, eqn11);

% Output A, b and c arrays
for i = 1 : 1
    A = [ 0, 0, 0, 0 ; a21, 0, 0, 0 ; a31, a32, 0, 0 ; a41, a42, a43, 0]
    b = [ b1 ; b2 ; b3 ; b4 ]
    c = [ 0 ; c2 ; c3 ; c4 ]
end
```

A =

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

b =

$$\begin{pmatrix} \frac{1}{6} \\ \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{6} \end{pmatrix}$$

c = 4×1

```
         0
    0.5000
    0.5000
    1.0000
```

So $a_{21} = \frac{1}{2}$, $a_{31} = 0$, $a_{32} = \frac{1}{2}$, $a_{41} = 0$, $a_{42} = 0$, $a_{43} = 1$, $b_1 = \frac{1}{6}$, $b_3 = \frac{1}{3}$ and $b_4 = \frac{1}{6}$ so the method is

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1), \\ k_3 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2), \\ k_4 &= f(t_n + h, y_n + hk_3), \end{aligned}$$

or as a Butcher tableau

$$\begin{array}{cccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ \frac{1}{2} & 0 & \frac{1}{2} & \\ 1 & 0 & 0 & 1 \\ & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

This fourth-order ERK is often referred to as *the* Runge-Kutta method or RK4 for short.

Example 5

Calculate the first two steps of the RK4 method to calculate the solution of the IVP in example 2 using a step length of $h = 0.5$.

$$y' = \sin^2(t)y, \qquad t \in [0, 5], \qquad y(0) = 1.$$

Step 1: $t_0 = 0$ and $y_0 = 1$

$$\begin{aligned} k_1 &= \sin^2(t_0)y_0 = 1\sin^2(0) = 0, \\ k_2 &= \sin^2(t_0 + \tfrac{1}{2}h)(y_0 + \tfrac{1}{2}hk_1) = \sin^2(0 + \tfrac{1}{2}(0.5))(1 + \tfrac{1}{2}(0.5)) = 0.0612, \\ k_3 &= \sin^2(t_0 + \tfrac{1}{2}h)(y_0 + \tfrac{1}{2}hk_2) = \sin^2(0 + \tfrac{1}{2}(0.5))(1 + \tfrac{1}{2}(0.5)(0.0612)) = 0.0631, \\ k_4 &= \sin^2(t_0 + h)(y_0 + hk_3) = \sin^2(0 + 0.5)(1 + 0.5(0.0621)) = 0.2443, \\ y_1 &= y_0 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) = 1 + \frac{0.5}{6}(0 + 2(0.0612) + 2(0.0621) + 0.2370) = 1.0411, \\ t_1 &= t_0 + h = 0 + 0.5 = 0.5. \end{aligned}$$

Step 2: $t_1 = 0.5$ and $y_1 = 1.0411$

$$\begin{aligned} k_1 &= \sin^2(t_1)y_1 = 1.0411\sin^2(0.5) = 0.2393, \\ k_2 &= \sin^2(t_1 + \tfrac{1}{2}h)(y_1 + \tfrac{1}{2}hk_1) = \sin^2(0.5 + \tfrac{1}{2}(0.5))(1.0411 + \tfrac{1}{2}(0.5)(0.2393)) = 0.5393, \\ k_3 &= \sin^2(t_1 + \tfrac{1}{2}h)(y_1 + \tfrac{1}{2}hk_2) = \sin^2(0.5 + \tfrac{1}{2}(0.5))(1.0411 + \tfrac{1}{2}(0.5)(0.5393)) = 0.6090, \\ k_4 &= \sin^2(t_1 + h)(y_1 + hk_3) = \sin^2(0.5 + 0.5)(1.0411 + (0.5)(0.6090)) = 1.1684, \\ y_2 &= y_1 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) = 1.0411 + \frac{0.5}{6}(0.2393 + 2(0.5393) + 2(0.6090) + 1.1684) = 1.3498, \\ t_2 &= t_1 + h = 0.5 + 0.5 = 1.0. \end{aligned}$$

Example 6

Calculate the solution of the IVP from example 2 over the domain $t \in [0, 5]$ using the RK4 method with a step length of $h = 0.5$.

The function called rk4 below calculates the solution to an IVP using the RK4 method using the same input arguments as the rk2 function defined earlier.

```
function [t, y] = rk4(f, tspan, y0, h)

% Calculates the solution to an IVP using the second-order Runge-Kutta method

% Initialise output arrays
nsteps = floor((tspan(2) - tspan(1)) / h);
t = zeros(nsteps, 1);
y = zeros(nsteps, 1);
t(1) = tspan(1);
y(1) = y0;

% Solver loop
for n = 1 : nsteps
    k1 = f(t(n), y(n));
    k2 = f(t(n) + 0.5 * h, y(n) + 0.5 * h * k1);
    k3 = f(t(n) + 0.5 * h, y(n) + 0.5 * h * k2);
    k4 = f(t(n) + h, y(n) + h * k3);
    y(n + 1) = y(n) + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4);
    t(n + 1) = t(n) + h;
end

end
```

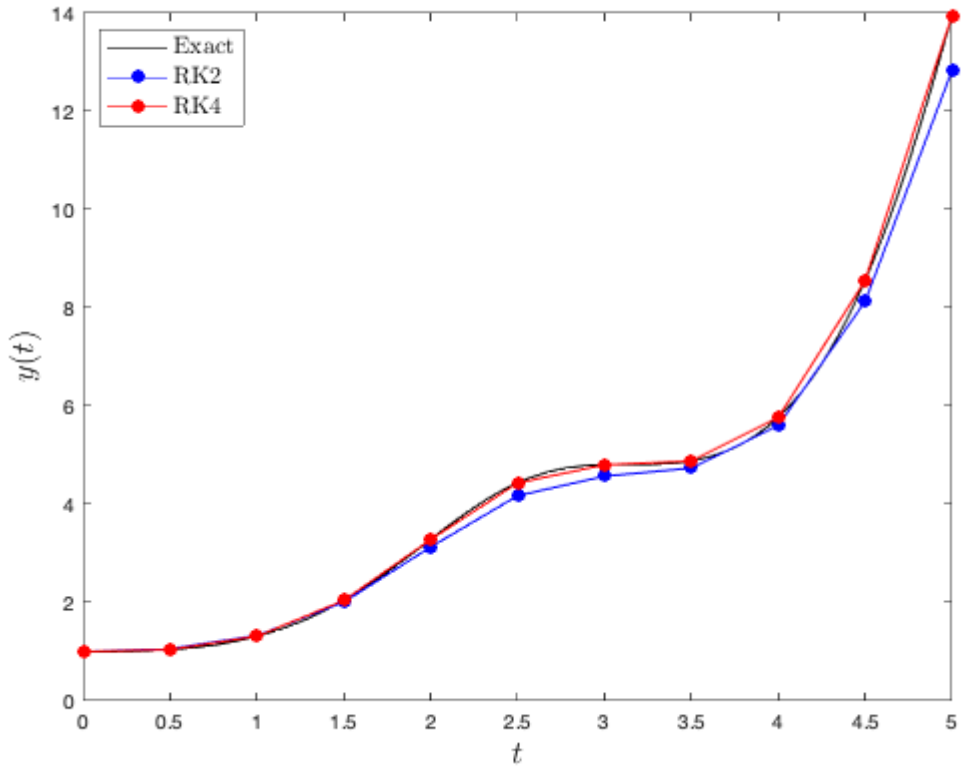
The program below invokes the RK4 method to solve the previously defined IVP.

```
% Solve the IVP using the RK2 and RK4 methods
[t1, y1] = rk2(f, tspan, y0, h);
[t2, y2] = rk4(f, tspan, y0, h);

% Output solution table
table = [' t     RK2     RK4    Exact \n', ...
    '---------------------------\n'];
for n = 1 : length(t)
    table = [table , sprintf('%3.2f %7.4f %7.4f %7.4f\n', ...
        t1(n), y1(n), y2(n), exact(t(n)))];
end
fprintf(table)
```

```
 t     RK2     RK4    Exact
---------------------------
0.00  1.0000  1.0000  1.0000
0.50  1.0575  1.0403  1.0404
1.00  1.3269  1.3133  1.3135
1.50  2.0088  2.0430  2.0436
2.00  3.1302  3.2830  3.2845
2.50  4.1734  4.4338  4.4359
3.00  4.5716  4.8032  4.8059
3.50  4.7364  4.8795  4.8830
4.00  5.6020  5.7656  5.7699
4.50  8.1257  8.5517  8.5589
5.00 12.8273 13.9431 13.9573
```

```
% Plot solutions
t3 = linspace(tspan(1), tspan(2), 100);
plot(t3, exact(t3), 'k-')
hold on
plot(t1, y1, 'bo-', 'markerfacecolor', 'b')
plot(t2, y2, 'ro-', 'markerfacecolor', 'r')
hold off
```

```
xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'RK2', 'RK4');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



The solution computed by the RK4 method shows very good agreement with the exact solutions. Whilst it is possible to derive higher-order Runge-Kutta methods, in practice the increased accuracy you get is not worth the additional computational cost. The RK4 method is considered the default ODE solver.

Summary

- Runge-Kutta methods are single step methods that use intermediate stage values to advance the solution over a small step of length h using known values of t_n and y_n to calculate $y_{n+1} = y(t_n + h)$.
- Runge-Kutta methods are commonly expressed in tabular form called a Butcher tableau.
- There are two types of Runge-Kutta methods: explicit and implicit.
- The coefficients of a Runge-Kutta methods are chosen to satisfy the order conditions which are derived by comparing the Taylor series expansion to the general form of the Runge-Kutta method to the equivalent Taylor series expansion of the first-order ODE $y' = f(t, y)$.

Exercises

1. Write the following Runge-Kutta method in a Butcher tableau.

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 4k_3 + k_4),\\
k_1 &= f(t_n, y_n),\\
k_2 &= f(t_n + \frac{1}{4}h, y_n + \frac{1}{4}hk_1),\\
k_3 &= f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2),\\
k_4 &= f(t_n + h, y_n + h(k_1 - 2k_2 + 2k_3)).
\end{aligned}
$$

2. Write out the equations for the following Runge-Kutta method.

$$
\begin{array}{cccc}
0 & & & \\
\frac{1}{4} & \frac{1}{4} & & \\
\frac{1}{2} & -\frac{1}{2} & 1 & \\
1 & \frac{1}{4} & 0 & \frac{3}{4} \\
 & 0 & \frac{4}{9} & \frac{1}{3} \quad \frac{2}{9}
\end{array}
$$

3. Derive an explicit second-order Runge-Kutta method such that $b_1 = \frac{1}{3}$.

4. Derive an explicit fourth-order Runge-Kutta method such that $b_1 = 0$ and $c_2 = \frac{1}{5}$.

5. Using pen and a calculator and working to 4 decimal places, apply your Runge-Kutta method derived in question 3 to solve the following IVP using a step length of $h = 0.4$

$$y' = ty, \qquad t \in [0, 2], \qquad y(0) = 1.$$

6. Repeat question 5 using the RK4 method (the RK4 method is derived in example 4).

7. Repeat question 5 using MATLAB to perform the calculations. The exact solution to this IVP is $y = e^{t^2/2}$. Produce a table comparing the numerical and exact solutions. Produce a plot of the number solutions and exact solutions on the same set of axes.

Implicit Runge-Kutta Methods

Learning outcomes

On successful completion of this chapter readers will be able to:

- Determine the order of an implicit Runge-Kutta method;
- Derive Gauss-Legendre, Radau, DIRK and SDIRK implicit Runge-Kutta methods;
- Apply implicit Runge-Kutta methods to solve initial value problems.

Implicit Runge-Kutta (IRK) methods and **Explicit Runge-Kutta** (ERK) methods are the two main types of the family of Runge-Kutta methods. ERK methods are straightforward to apply to solve an ordinary differential equation but are not suitable for **stiff** systems. This is why we need to also consider IRK methods.

The general form of an IRK method to solve a first-order ODE $y' = f(t, y)$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i,$$

$$k_i = f(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j).$$

Here $y_n = y(t_n)$ is the solution at a discrete point in the domain t_n; $y_{n+1} = y(t_n + h)$ is the solution at some small step h along the domain; a_{ij}, b_i and c_i are coefficients that define a particular method and k_i are intermediate **stage values** used to calculate y_{n+1}.

It is common to represent the a_{ij}, b_i and c_i coefficients in a **Butcher tableau**

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

Note that Butcher tableaus are usually represented with partition lines separating the a_{ij}, b_i and c_i values (see examples). MATLAB's LaTeX renderer does not permit this for some reason.

The A matrix of an IRK method has non-zero elements in the upper triangular region and on the main diagonal whereas the A matrix for an ERK method only has non-zero elements in the lower triangular region.

Determining the order of an implicit Runge-Kutta method

One of the differences between IRK and ERK methods is that an IRK method can achieve the same accuracy as an ERK method but using fewer stages. To determine the order of an IRK method we need to consider the **order conditions** that govern the values of A, $\mathbf{b}$ and $\mathbf{c}$ which are

$$B(K): \quad \sum_{i=1}^{s} b_i c_i^{j-1} = \frac{1}{j}, \qquad j = 1, 2, \ldots, k, \tag{1}$$

$$C(k): \quad \sum_{j=1}^{s} a_{ij} c_j^{\ell-1} = \frac{1}{\ell} c_i^{\ell}, \qquad i = 1, 2, \ldots, s, \quad \ell = 1, 2, \ldots, k, \tag{2}$$

$$D(k)): \quad \sum_{i=1}^{s} b_i c_i^{\ell-1} a_{ij} = \frac{1}{\ell} b_j (1 - c_j^{\ell}), \quad j = 1, 2, \ldots, s, \quad \ell = 1, 2, \ldots, k. \tag{3}$$

If $G(k)$ represents the fact that a given IKR method has order k then it can be shown that

$$B(k) \text{ and } C(\lfloor k/2 \rfloor) \text{ and } D(\lfloor k/2 \rfloor) \Longrightarrow G(k)$$

So to determine the order of an IRK method we need to find the highest value of k for which equation (1) is satisfied and which equations (2) and (3) are satisfied for $\lfloor k/2 \rfloor$ (i.e. the integer part of $k/2$).

Example 1

Determine the order of the following IRK method

$$\begin{array}{ccc} \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ & \frac{1}{2} & \frac{1}{2} \end{array}$$

Checking the $B(k)$ order condition:

$$j = 1: \quad LHS = b_1 c_1^0 + b_2 c_2^0 = \frac{1}{2}(1) + \frac{1}{2}(1) = 1, \qquad RHS = 1,$$

$$j = 2: \quad LHS = b_1 c_1^1 + b_2 c_2^1 = \frac{1}{2}\left(\frac{1}{4}\right) + \frac{1}{2}\left(\frac{3}{4}\right) = \frac{1}{2}, \qquad RHS = \frac{1}{2},$$

$$j = 3: \quad LHS = b_1 c_1^2 + b_2 c_2^2 = \frac{1}{2}\left(\frac{1}{4}\right)^2 + \frac{1}{2}\left(\frac{3}{4}\right)^2 = \frac{5}{16}, \qquad RHS = \frac{1}{3}.$$

So the $B(k)$ order condition is satisfied up to $k = j = 2$. Now we need to check whether $C(k)$ order condition is satisfied up to $k = 1$.

$$\ell = 1, \quad i = 1, \quad LHS = a_{11} c_1^0 + a_{12} c_2^0 = \frac{1}{4}(1) + 0(1) = \frac{1}{4}, \quad RHS = \frac{1}{\ell} c_1^{\ell} = \frac{1}{4},$$

$$\ell = 1, \quad i = 2, \quad LHS = a_{21} c_1^0 + a_{22} c_2^0 = \frac{1}{2}(1) + \frac{1}{4}(1) = \frac{3}{4}, \quad RHS = \frac{1}{\ell} c_2^{\ell} = \frac{3}{4}.$$

So the $C(k)$ order condition is satisfied up to $k = 1$. Now we need to check whether the $D(k)$ order condition is satisfied up to $k = 1$

$$\ell = 1, \quad j = 1, \quad LHS = b_1 c_1^0 a_{11} + b_2 c_2^0 a_{21} = \frac{1}{2}(1)\left(\frac{1}{4}\right) + \frac{1}{2}(1)\left(\frac{1}{2}\right) = \frac{3}{8},$$
$$RHS = \frac{1}{\ell} b_1 (1 - c_1^\ell) = \frac{1}{2}\left(1 - \frac{1}{4}\right) = \frac{3}{8},$$
$$\ell = 1, \quad j = 2, \quad LHS = b_1 c_1^0 a_{12} + b_2 c_2^0 a_{22} = \frac{1}{2}(1)(0) + \frac{1}{2}(1)\left(\frac{1}{4}\right) = \frac{1}{8},$$
$$RHS = \frac{1}{\ell} b_2 (1 - c_2^\ell) = \frac{1}{2}\left(1 - \frac{3}{4}\right) = \frac{1}{8}.$$

So the $D(k)$ order condition is satisfied up to $k = 1$. Therefore since $B(2)$, $C(1)$ and $D(1)$ are all satisfied this IRK method is order 2.

Deriving implicit Runge-Kutta methods

Gauss-Legendre methods

Gauss-Legendre IRK methods are a family of methods that are derived using Gauss-Legendre quadrature. An s-stage Gauss-Legendre method has order $k = 2s$. They are derived using Legendre polynomials which can take the form

$$P_n(t) = \sum_{k=0}^{n} \binom{n}{k}\binom{n+k}{k}(t-1)^k, \qquad (4)$$

where $\binom{n}{k}$ is the Binomial coefficient. The values of the c_i coefficients are the roots of $P_s(t)$, the values of the b_i coefficients are chosen to satisfy the $B(k)$ condition and the a_{ij} coefficients are chosen to satisfy the $C(\lfloor k/2 \rfloor)$ condition.

Example 2

Derive a fourth-order Gauss-Legendre method.

A fourth-order Gauss-Legendre method will have $s = 2$ stages and the c_i coefficients are chosen to satisfy $0 = P_2(t)$

$$\begin{aligned} 0 &= \binom{2}{0}\binom{2}{0}(t-1)^0 + \binom{2}{1}\binom{3}{1}(t-1)^1 + \binom{2}{2}\binom{4}{2}(t-1)^2 \\ &= 1 + 6t - 6 + 6t^2 - 12t + 6 \\ &= 6t^2 - 6t + 1 \end{aligned}$$

therefore $c_1 = \frac{1}{2} - \frac{\sqrt{3}}{6}$ and $c_2 = \frac{1}{2} + \frac{\sqrt{3}}{6}$. The b_i and a_{ij} coefficients are chosen to satisfy the $B(4)$ and $C(2)$ order conditions respectively

$$
\begin{aligned}
b_1 + b_2 &= 1, \\
b_1 c_1 + b_2 c_2 &= \frac{1}{2}, \\
a_{11} + a_{12} &= c_1, \\
a_{21} + a_{22} &= c_2, \\
a_{11} c_1 + a_{12} c_2 &= \frac{1}{2} c_1^2, \\
a_{21} c_1 + a_{22} c_2 &= \frac{1}{2} c_2^2.
\end{aligned}
$$

The code below uses the solve command to solve these order conditions for the coefficients a_{ij}, b_i and c_i.

```
clear

% Define symbolic variables
syms t a11 a12 a21 a22 b1 b2 c1 c2

% Solve for c1 and c2
c = solve(6 * t^2 - 6 * t + 1 == 0);
c1 = c(1);
c2 = c(2);

% Define order conditions
eqn1 = b1 + b2 == 1;
eqn2 = b1 * c1 + b2 * c2 == 1/2;
eqn3 = a11 + a12 == c1;
eqn4 = a21 + a22 == c2;
eqn5 = a11 * c1 + a12 * c2 == 1/2 * c1^2;
eqn6 = a21 * c1 + a22 * c2 == 1/2 * c2^2;

% Solve the order conditions
[a11, a12, a21, a22, b1, b2] = solve(eqn1, eqn2, eqn3, eqn4, eqn5, eqn6);
a12 = simplify(a12);

% Output A, b and c arrays
for i = 1 : 1
    A = [ a11, a12 ; a21, a22 ]
    b = [ b1 ; b2 ]
    c = [ c1 ; c2 ]
end
```

A =

$$
\begin{pmatrix}
\frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\
\frac{\sqrt{3}}{6} + \frac{1}{4} & \frac{1}{4}
\end{pmatrix}
$$

b =

$$
\begin{pmatrix}
\frac{1}{2} \\
\frac{1}{2}
\end{pmatrix}
$$

c =

$$\begin{pmatrix} \frac{1}{2} - \frac{\sqrt{3}}{6} \\ \frac{\sqrt{3}}{6} + \frac{1}{2} \end{pmatrix}$$

Radau methods

Gauss-Legendre methods give us maximal order for the number of stages, however sometimes it is better the sacrifice order to gain better stability properties. An s-stage **Radau** IRK method has order $k = 2s - 1$ and is A-stable. There are two types of Radau methods: Radau IA and Radau IIA.

Radau IA:

Let $c_1 = 0$ and the other c_i coefficients are the roots of

$$0 = P_s(t) + P_{s-1}(t).$$

The coefficients b_i are chosen to satisfy the $B(2s)$ order condition and the coefficients a_{ij} are chosen to satisfy the $C(s)$ order condition.

Radau IIA

Let $c_2 = 1$ and the other c_i coefficients are the roots of

$$0 = P_s(t) - P_{s-1}(t).$$

The coefficients b_i are chosen to satisfy the $B(2s)$ order condition and the coefficients a_{ij} are chosen to satisfy the $C(s)$ order condition.

Example 3

Derive a third-order Radau IA method.

A $k = 3$ order Radau IA method will have $s = 2$ stages. Let $c_1 = 0$ then the value of c_2 is the root of $0 = P_2(t) + P_1(t)$

$$0 = (6t^2 - 6t + 1) + (2t - 1) = 6t^2 - 4t,$$

which is $c_2 = \frac{2}{3}$. The values of b_i and a_{ij} need to satisfy the $B(4)$ and $C(2)$ order conditions respectively with $k = 2$ (same as in example 2).

```
clear

% Define symbolic variables
syms t a11 a12 a21 a22 b1 b2 c1 c2

% Calculate c2
c = solve(6 * t^2 - 4 * t == 0);
c1 = 0;
c2 = c(2);

% Define order conditions
```

```
eqn1 = b1 + b2 == 1;
eqn2 = b1 * c1 + b2 * c2 == 1/2;
eqn3 = a11 + a12 == c1;
eqn4 = a21 + a22 == c2;
eqn5 = a11 * c1 + a12 * c2 == 1/2 * c1^2;
eqn6 = a21 * c1 + a22 * c2 == 1/2 * c2^2;

% Solve system
[a11, a12, a21, a22, b1, b2] = solve(eqn1, eqn2, eqn3, eqn4, eqn5, eqn6);

% Output A, b and c arrays
for i = 1 : 1
    A = [ a11, a12 ; a21, a22 ]
    b = [ b1 ; b2 ]
    c = [ c1 ; c2 ]
end
```

A =

$$\begin{pmatrix} 0 & 0 \\ \frac{1}{3} & \frac{1}{3} \end{pmatrix}$$

b =

$$\begin{pmatrix} \frac{1}{4} \\ \frac{3}{4} \end{pmatrix}$$

c =

$$\begin{pmatrix} 0 \\ \frac{2}{3} \end{pmatrix}$$

Diagonally implicit Runge-Kutta methods

Diagonally Implicit Runge-Kutta (DIRK) methods are where the A matrix is lower triangular with non-zero elements on the main diagonal, i.e.,

$$A = \begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{s1} & a_{s2} & \cdots & a_{ss} \end{pmatrix}$$

The advantage of DIRK methods is that although the stage values $k_1, k_2, \ldots, k_s$ are defined using implicit equations, the solutions to these can be obtained sequentially since the equation for k_i does not include $k_{i+1}, k_{i+2}, \ldots, k_s$.

The coefficients of a kth-order DIRK method are chosen to satisfy the $B(k)$ and $C(\lfloor k/2 \rfloor)$ order conditions along with

$$\mathbf{b}^T \cdot A \cdot \mathbf{c} = \frac{1}{k!}.$$

Example 4

Derive a 2-stage third-order DIRK method.

Since $k = 3$ the order conditions are (remembering that $a_{12} = 0$)

$$\begin{aligned} b_1 + b_2 &= 1, \\ b_1 c_1 + b_2 c_2 &= \frac{1}{2}, \\ b_1 c_1^2 + b_2 c_2^2 &= \frac{1}{3}, \\ a_{11} &= c_1, \\ a_{21} + a_{22} &= c_2, \\ b_1 a_{11} c_1 + b_2 (a_{21} c_1 + a_{22} c_2) &= \frac{1}{6} \end{aligned}$$

Here we have a system of 6 equations in 7 unknowns. Choosing $c_1 = \frac{1}{4}$ and solving for the other coefficients.

```
clear

% Define symbolic variables
syms a11 a21 a22 b1 b2 c1 c2
c1 = 1/4;

% Define order conditions
eqn1 = b1 + b2 == 1;
eqn2 = b1 * c1 + b2 * c2 == 1/2;
eqn3 = b1 * c1^2 + b2 * c2^2 == 1/3;
eqn4 = a11 == c1;
eqn5 = a21 + a22 == c2;
eqn6 = b1 * a11 * c1 + b2 * (a21 * c1 + a22 * c2) == 1/6;

% Solve system
[a11, a21, a22, b1, b2, c2] = solve(eqn1, eqn2, eqn3, eqn4, eqn5, eqn6);

% Output A, b and c arrays
for i = 1 : 1
    A = [ a11, 0 ; a21, a22 ]
    b = [ b1 ; b2 ]
    c = [ c1 ; c2 ]
end
```

A =

$$\begin{pmatrix} \frac{1}{4} & 0 \\ \frac{2}{3} & \frac{1}{6} \end{pmatrix}$$

b =

$$\begin{pmatrix} \frac{4}{7} \\ \frac{3}{7} \end{pmatrix}$$

c =

$$\begin{pmatrix} \frac{1}{4} \\ \frac{5}{6} \end{pmatrix}$$

Singly diagonally implicit Runge-Kutta methods

Singly Diagonally Implicit Runge-Kutta (SDIRK) methods are a variation on DIRK method with the additional condition that the elements on the main diagonal have the same value (i.e., $a_{ii} = c_1$ by the row sum condition on the first row of A).

$$A = \begin{pmatrix} c_1 & & & \\ a_{21} & c_1 & & \\ \vdots & \vdots & \ddots & \\ a_{s1} & a_{s2} & \cdots & c_1 \end{pmatrix}$$

The advantage that SDIRK methods is that they can be A-stable for certain values of c_1.

The derivation of an kth-order SDIRK method uses the $B(k)$, $C(\lfloor k/2 \rfloor)$ and $D(\lfloor k/2 \rfloor)$ order conditions.

Example 5

Derive two-stage third-order SDIRK method.

Since $a_{11} = a_{22} = c_1$ the order conditions used in example 4 become

$$\begin{aligned} b_1 + b_2 &= 1, \\ b_1 c_1 + b_2 c_2 &= \frac{1}{2}, \\ b_1 c_1^2 + b_2 c_2^2 &= \frac{1}{3}, \\ a_{21} + c_1 &= c_2, \\ b_1 c_1^2 + b_2 (a_{21} c_1 + c_1 c_2) &= \frac{1}{6} \end{aligned}$$

Here we have 5 equations in 5 unknowns. Solving these order conditions:

```
clear

% Define symbolic variables
syms a11 a21 b1 b2 c1 c2

% Define order conditions
eqn1 = b1 + b2 == 1;
eqn2 = b1 * c1 + b2 * c2 == 1/2;
```

```
eqn3 = b1 * c1^2 + b2 * c2^2 == 1/3;
eqn4 = a21 + c1 == c2;
eqn5 = b1 * c1^2 + b2 * (a21 * c1 + c1 * c2) == 1/6;

% Solve system
[a21, b1, b2, c1, c2] = solve(eqn1, eqn2, eqn3, eqn4, eqn5);

% Output A, b and c arrays
for i = 1 : length(b1)
    A = [ c1(i), 0 ; a21(i), c1(i) ]
    b = [ b1(i) ; b2(i) ]
    c = [ c1(i) ; c2(i) ]
end
```

A =

$$\begin{pmatrix} \frac{\sqrt{3}}{6}+\frac{1}{2} & 0 \\ -\frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{6}+\frac{1}{2} \end{pmatrix}$$

b =

$$\begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

c =

$$\begin{pmatrix} \frac{\sqrt{3}}{6}+\frac{1}{2} \\ \frac{1}{2}-\frac{\sqrt{3}}{6} \end{pmatrix}$$

A =

$$\begin{pmatrix} \frac{1}{2}-\frac{\sqrt{3}}{6} & 0 \\ \frac{\sqrt{3}}{3} & \frac{1}{2}-\frac{\sqrt{3}}{6} \end{pmatrix}$$

b =

$$\begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

c =

$$\begin{pmatrix} \frac{1}{2}-\frac{\sqrt{3}}{6} \\ \frac{\sqrt{3}}{6}+\frac{1}{2} \end{pmatrix}$$

Note that here we have two SDIRK methods alternating the sign of the $\sqrt{3}$ term.

Implementation of implicit Runge-Kutta methods

Recall that the general form of an IRK method is

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i,$$

$$k_i = f(t_n + c_i h, y_n + h\sum_{j=1}^{s} a_{ij} k_j).$$

Let $Y_i = y_n + h\sum_{j=1}^{s} a_{ij}k_j$ then

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i f(t_n + c_i h, Y_i),$$

$$Y_i = y_n + h\sum_{j=1}^{s} a_{ij} f(t_n + c_j h, Y_j).$$

Expanding the summation in the expression for Y_i givens

$$\begin{aligned}
Y_1 &= y_n + h(a_{11}f(t_n + c_1 h, f(t_n + c_1 h, Y_1) + a_{12}f(t_n + c_2 h, Y_2) + \cdots + a_{1s}f(t_n + c_s h, Y_s)),\\
Y_2 &= y_n + h(a_{21}f(t_n + c_1 h, f(t_n + c_1 h, Y_1) + a_{22}f(t_n + c_2 h, Y_2) + \cdots + a_{2s}f(t_n + c_s h, Y_s)),\\
&\vdots\\
Y_s &= y_n + h(a_{s1}f(t_n + c_1 h, f(t_n + c_1 h, Y_1) + a_{s2}f(t_n + c_2 h, Y_2) + \cdots + a_{ss}f(t_n + c_s h, Y_s)),
\end{aligned}$$

Let $Y = (Y_1, Y_2, \ldots, Y_s)^T$ then we can write

$$Y = y_n + hA\, f(t_n + \mathbf{c}h, Y),$$

and

$$y_{n+1} = y_n + h\mathbf{b}^T f(t_n + \mathbf{c}h, Y). \qquad (5)$$

Since the equation for the stage values Y is an implicit equation we need to use an iterative method to solve this. The simplest iterative method is the Jacobi method. Let Y_k denote an estimate of Y and Y^{k+1} denote an improved estimate. Using non-zero starting values for Y_0 the Jacobi method applied to calculate the stage values of an IRK is

$$Y^{k+1} = y_n + hA\, f(t_n + \mathbf{c}h, Y^k). \qquad (6)$$

The method is iterated until the largest absolute difference between two successive iterations of the method is less than some small number tol, i.e.,

$$\max |Y^{k+1} - Y^k| < tol.$$

Note that the Jacobi method is computationally inefficient and in practice Newton's method is most commonly used to calculate Y. This requires calculation of a Jacobian matrix and the solution to a linear system of equations and is outside of the scope of this course.

Example 6

Calculate the first step of the following IRK method

$$\begin{array}{ccc} 0 & \frac{1}{4} & -\frac{1}{4} \\ \frac{2}{3} & \frac{1}{4} & \frac{5}{12} \\ & \frac{1}{4} & \frac{3}{4} \end{array}$$

used to solve the IVP

$$y' = \sin^2(t)y, \qquad t \in [0,5], \qquad y(0) = 1,$$

using a step length of $h = 0.5$.

Since $t_0 = 0$ and $y_0 = 1$ we need to iterate equation (6) to solve for the stage values Y. Using $Y^0 = (1,1)^T$:

$$Y^1 = y_0 + hAf(t_0 + \mathbf{c}h, Y^0) = 1 + 0.5\begin{pmatrix} \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{5}{12} \end{pmatrix}\begin{pmatrix} \sin^2(0+0(0.5))(1) \\ \sin^2\left(0+\frac{2}{3}(0.5)\right)(1) \end{pmatrix} = \begin{pmatrix} 0.9866 \\ 1.0223 \end{pmatrix},$$

$$Y^2 = y_0 + hAf(t_0 + \mathbf{c}h, Y^1) = 1 + 0.5\begin{pmatrix} \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{5}{12} \end{pmatrix}\begin{pmatrix} \sin^2(0+0(0.5))(0.9866) \\ \sin^2\left(0+\frac{2}{3}(0.5)\right)(1.0223) \end{pmatrix} = \begin{pmatrix} 0.9863 \\ 1.0228 \end{pmatrix},$$

$$Y^3 = y_0 + hAf(t_0 + \mathbf{c}h, Y^2) = 1 + 0.5\begin{pmatrix} \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{5}{12} \end{pmatrix}\begin{pmatrix} \sin^2(0+0(0.5))(0.9863) \\ \sin^2\left(0+\frac{2}{3}(0.5)\right)(1.0228) \end{pmatrix} = \begin{pmatrix} 0.9863 \\ 1.0228 \end{pmatrix}.$$

Here the values of Y^2 and Y^3 agree to 4 decimal places. Substituting these into equation (5)

$$y_1 = y_0 + h\mathbf{b}^T f(t_0 + \mathbf{c}h, Y) = 1 + 0.5\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix}\begin{pmatrix} \sin^2(0+0(0.5))(1) \\ \sin^2\left(0+\frac{2}{3}(0.5)\right)(1.0228) \end{pmatrix} = 1.0411.$$

Example 7

Calculate the solution of the IVP from example 6 over the whole domain $t \in [0,5]$ using the given IRK method with a step length of $h = 0.5$ and compare it to the exact solution given by

$$y = e^{(t-\sin(t)\cos(t))/2}.$$

The function called irk below calculates the solution to an IVP using an IRK method. The input arguments are `f` is the name of the function the defines the ODE, `tspan` is a two-element array containing the lower and upper bounds of the t domain, `y0` is the initial value of y at the lower bound, `h` is the step length and `A`, `b` and `c` are the coefficients that define an IRK method.

```
function [t, y] = irk(f, tspan, y0, h, A, b, c)

% Calculates the solution to an initial value problem using an implicit
% Runge-Kutta method defined by A, b and c
```

```
% Initialise output arrays
nsteps = floor((tspan(2) - tspan(1))/ h);
t = zeros(nsteps + 1);
y = zeros(nsteps + 1);
t(1) = tspan(1);
y(1) = y0;

% Loop through steps
for n = 1 : nsteps

    % Solve for Y values using the Jacobi method
    Y = ones(size(b));
    for k = 1 : 100
        Ynew = y(n) + h * A * f(t(n) + c * h, Y);

        % Check for convergence
        if max(abs(Ynew - Y)) < 1e-6
            break
        end

        % Update Y
        Y = Ynew;
    end

    % Calculate the solution at the next step
    y(n+1) = y(n) + h * b' * f(t(n) + c * h, Y);
    t(n+1) = t(n) + h;

end

end
```

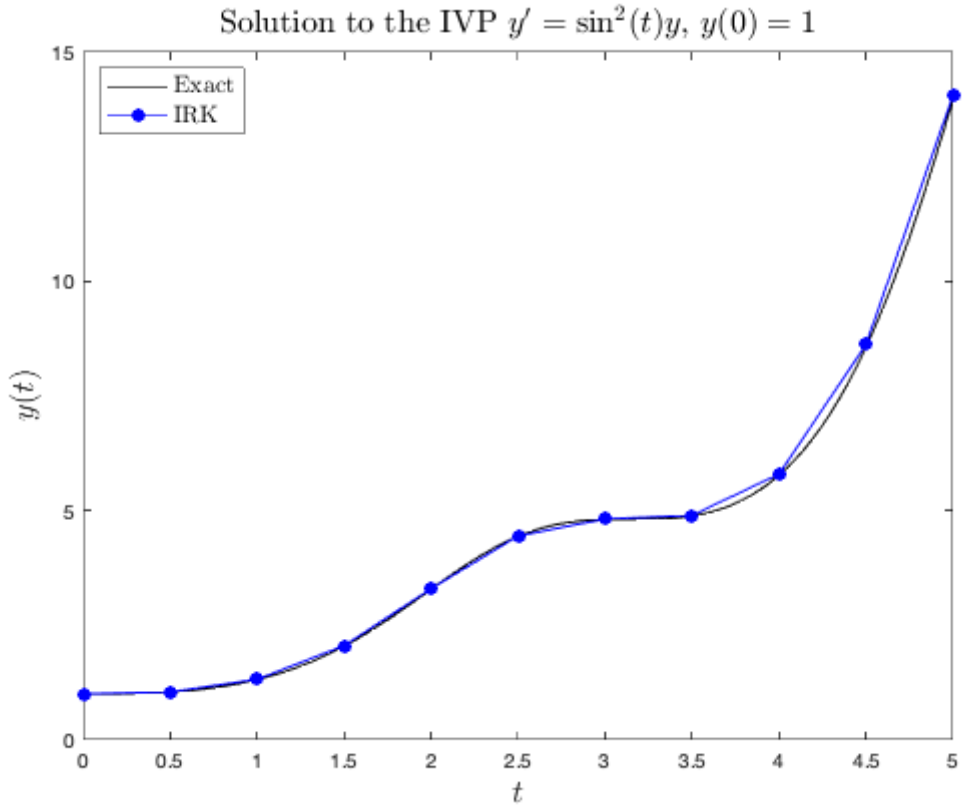
The program below invokes the IRK method given here to solve this IVP.

```
clear

% Define ODE function
f = @(t, y) sin(t).^2 .* y;

% Define exact solution
exact = @(t) exp(0.5 * (t - sin(t) .* cos(t)));

% Define IRK method
A = [ 1/4, -1/4 ; 1/4, 5/12 ];
b = [ 1/4 ; 3/4 ];
c = [ 0 ; 2/3 ];

% Define IVP parameters
tspan = [ 0, 5 ];
y0 = 1;
h = 0.5;

% Invoke RK2 method to solve IVP
[t, y] = irk(f, tspan, y0, h, A, b, c);

% Output solution table
table = sprintf(' t     IRK    Exact \n--------------------\n');
for n = 1 : length(t)
    table = [table , sprintf('%3.2f %7.4f %7.4f\n', t(n), y(n), exact(t(n)))];
end
```

```
fprintf(table)
```

```
 t     IRK    Exact
--------------------
0.00  1.0000  1.0000
0.50  1.0411  1.0404
1.00  1.3174  1.3135
1.50  2.0535  2.0436
2.00  3.2955  3.2845
2.50  4.4465  4.4359
3.00  4.8160  4.8059
3.50  4.8943  4.8830
4.00  5.7944  5.7699
4.50  8.6166  8.5589
5.00 14.0405 13.9573
```

```
% Plot solutions
t1 = linspace(tspan(1), tspan(2), 100);
plot(t1, exact(t1), 'k-')
hold on
plot(t, y, 'bo-', 'markerfacecolor', 'blue')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
title("Solution to the IVP $y'=\sin^2(t)y$, $y(0)=1$", 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'IRK');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



We can see here that the solution computed using this IRK method (which is of order 3) closely matches the exact solution.

Summary

- The stage values in an Implicit Runge-Kutta (IRK) method are defined using implicit equations.
- The A matrix of an IRK method has non-zero elements on the main diagonal or upper triangular region.
- IRK methods use fewer stages to achieve the same accuracy as Explicit Runge-Kutta (ERK) methods.
- The derivation of IRK methods involves choosing values of a_{ij}, b_i and c_i to satisfy some order conditions.
- IRK methods have better stability properties than ERK methods meaning they can be used to solve stiff ODEs with which ERK methods are unsuitable.
- IRK methods require an iterative method such as the Jacobi method to calculate the stage values.
- IRK methods are more computationally expensive when applied to non-stiff methods than ERK methods.

Exercises

1. Show that the Radau IA derived in example 3 is a third-order method.

2. Determine the order of the following IRK method

$$\begin{array}{ccc|c} 0 & 0 & 0 & \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ 1 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

3. Derive a third-order Radau IIA method.

4. Derive a two-stage second-order SDIRK method with $c_1 = \frac{1}{4}$.

5. Using pen and a calculator and working to 4 decimal places, calculate the first step of your IRK method derived in question 4 to solve the following IVP using a step length of $h = 0.4$

$$y' = ty, \qquad t \in [0, 2], \qquad y(0) = 1.$$

6. The exact solution to the IVP in question 5 is $y = e^{t^2/2}$. Using MATLAB repeat question 5 for the full domain and produce a table comparing the numerical and exact solutions and plot the numerical solutions and exact solutions on the same set of axes.

Stability of Runge-Kutta Methods

Learning outcomes

On successful completion of this chapter readers will be able to:

- Understand the concept of local and global truncation errors and what it means for a method to be considered stable.
- Determine the stability function of explicit and implicit Runge-Kutta method.
- Plot the region of absolute stability.
- Determine whether an implicit Runge-Kutta method is A-stable or not.

The derivation of a numerical method to solve an ODE involves omitting the higher-order terms from the Taylor series known as **truncating** the Taylor series. In doing this we introduce an error at each step of the method, and it is important to be able to analyse the extent to which these errors affect the numerical solutions. In the majority of cases the exact solution of an ODE is unknown (if it was known we would not need to use a numerical method to solve it) so we cannot determine the values of the errors but we can examine their behaviour through each step of the method.

To do this it is necessary to introduce terminology used to describe the truncation errors.

Definition (local truncation error)

Let y_n be a numerical approximation of the exact solution $\bar{y}_n$ for some step n of a method then the **local truncation error** is

$$\tau_n = y_n - \bar{y}_n,$$

Definition (global truncation error)

The **global truncation error** is the accumulation of the local truncation errors up to the current step

$$E_n = \sum_{i=0}^{n} \tau_i.$$

If $|\tau_{i+1} - \tau_i| > 1$ (i.e., the local truncation errors grow from one step to the next) then $E_n \to \infty$ as n gets large and the method is said to be **unstable** and unusable. So for a method to be considered **stable** we need the growth in the local truncation errors to remain bounded which leads to the definition of numerical stability.

Definition (stability)

If τ_n is the local truncation error of a numerical method for solving a differential equation, then the method is considered **stable** if

$$|\tau_{n+1} - \tau_n| \leq 1,$$

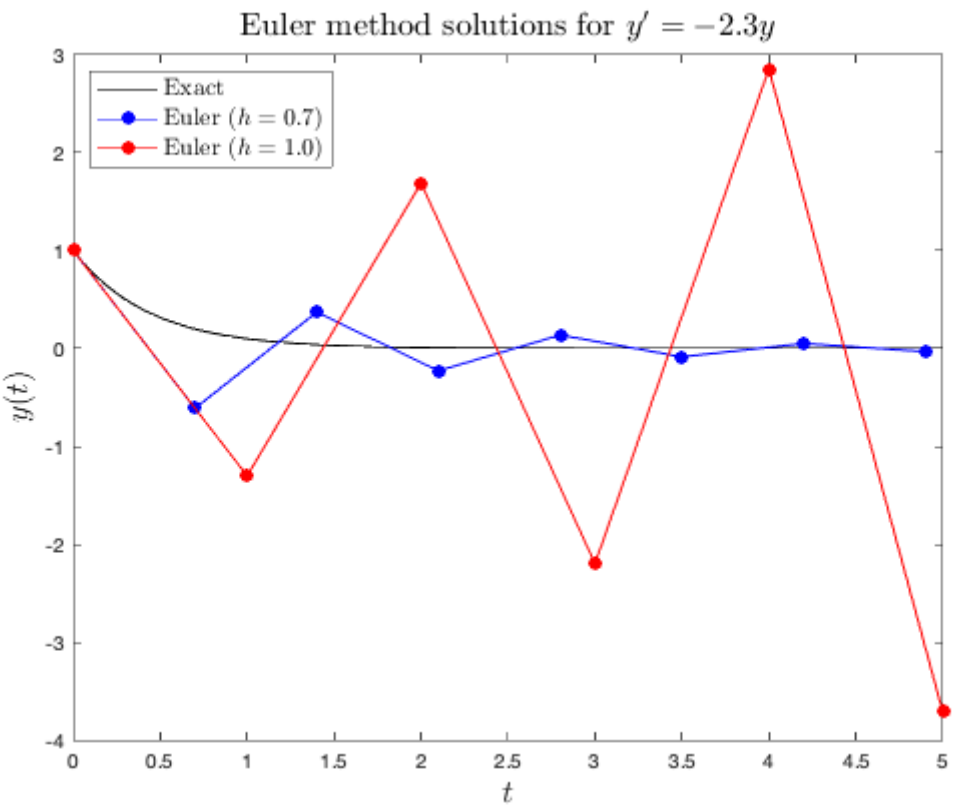
for all steps of the method.

Since the Taylor series truncation error $O(h^n)$ is some function of the step length h, as h increases then so does the truncation error and the more likely a method is to be unstable.

Example 1

Consider the Euler method when applied to solve the ODE $y' = -2.3y$ over the domain $t \in [0, 5]$ with an initial condition $y(0) = 1$. This ODE has the exact solution $y = e^{-2.3t}$. The code below compares the Euler method solutions using two steps lengths of $h = 0.7$ and $h = 1$ to the exact solution.

```
clear

% Define ODE function
f = @(t, y) -2.3 .* y;

% Define exact solution
exact_sol = @(t) exp(-2.3 .* t);

% Define IVP parameters
tspan = [ 0, 5 ];
y0 = 1;
h1 = 0.7;
h2 = 1.0;

% Solve IVP using the Euler method
[t1, y1] = euler(f, tspan, y0, h1);
[t2, y2] = euler(f, tspan, y0, h2);

% Plot solution
t3 = linspace(tspan(2), tspan(1), 100);
plot(t3, exact_sol(t3), 'k-')
hold on
plot(t1, y1, 'bo-', 'MarkerFaceColor', 'b')
plot(t2, y2, 'ro-', 'MarkerFaceColor', 'r')
hold off

xlabel('$t$', 'fontsize', 16, 'Interpreter', 'latex')
ylabel('$y(t)$', 'FontSize', 16, 'Interpreter', 'latex')
title("Euler method solutions for $y'=-2.3y$", 'FontSize', 16, 'Interpreter', 'latex')
leg = legend('Exact', 'Euler ($h=0.7$)', 'Euler ($h=1.0$)');
set(leg, 'Location', 'Northwest', 'FontSize', 12, 'Interpreter', 'latex')
```



Here we can see that the solution using $h = 0.7$ remains stable whereas the solution using $h = 1.0$ is diverging and unstable.

Stability Functions

To examine the behaviour of the local truncation errors as we step through a method we use the test ODE $y' = \lambda y$. As the values of y_{n+1} are updated using the values of y_n so are the values of τ_{n+1} using τ_n by the same method. This allows us to define a **stability function** for a method.

Definition (stability function)

The **stability function** of a method, $R(z)$ is the rate of growth over a single step of the method when applied to calculate the solution of an ODE of the form $y' = \lambda y$ where $z = h\lambda$ and h is the step size, i.e.,

$$y_{n+1} = R(z)y_n.$$

Example 2

Determine the stability function for the Euler method.

If the Euler method is used to solve and ODE of the form $y' = \lambda y$ then the solution will be updated over one step using

$$y_{n+1} = y_n + hf(t_n, y_n),$$

then the local truncation errors will also update in the same step by

$$\tau_{n+1} = \tau_n + h(t_n, \tau_n).$$

Applying the Euler method to the test ODE we have

$$y_{n+1} = y_n + h\lambda y_n.$$

Let $z = h\lambda$ then

$$\begin{aligned} y_{n+1} &= y_n + zy_n \\ &= (1+z)y_n. \end{aligned}$$

So the stability function of the Euler method is $R(z) = 1 + z$.

Absolute stability

We have seen that a necessary condition for stability of a method is that the local truncation errors must not grow from one step to the next. A method satisfying this basic condition is considered to be **absolutely stable**. Since the stability function $R(z)$ is expressed using $z = h\lambda$ then a method may be stable for some value of z and unstable for others. This provides the definition for absolute stability.

Definition (absolute stability)

A method is considered to be **absolutely stable** if $|R(z) \leq 1|$ for $z \in \mathbb{C}$.

Of course we require our methods to be stable, so it is useful to know for what values of zz we have a stable method. This gives the definition of the **region of absolute stability**.

Definition (region of absolute stability)

The **region of absolute stability** is the set of $z \in \mathbb{C}$ for which a method is absolutely stable

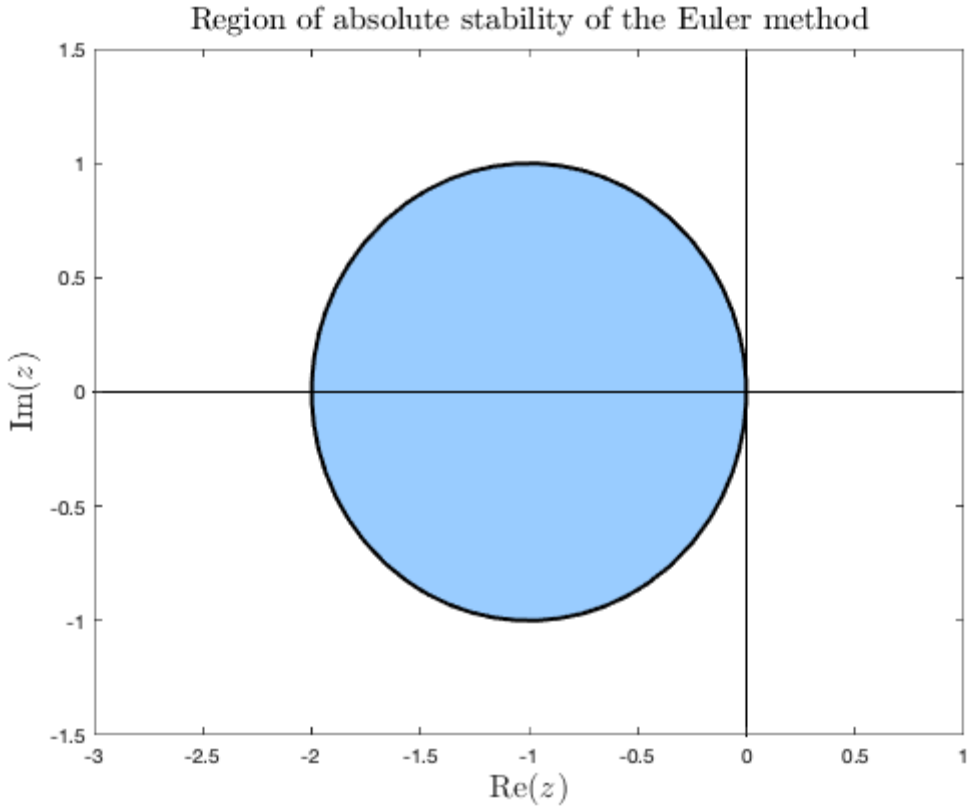
$$\text{region of absolute stability} = \{z : z \in \mathbb{C}, |R(z)| \leq 1\}.$$

Example 3

The stability function for the Euler method is $R(z) = 1 + z$. The code below generates a set of points in the complex plane $z = x + iy$ and plots the contour where $|R(z)| = 1$ which represents the boundary of the stability region of the Euler method.

```
clear

% Generate z values
x = linspace(-5, 5, 100);
y = linspace(-5, 5, 100);
[X, Y] = meshgrid(x, y);
```

```
Z = X + Y * 1i;

% Define the stability function for the euler method
R = 1 + Z;

% Plot the region of absolute stability
contourf(X, Y, abs(R), [0 1], 'linewidth', 2)
hold on
plot([ -10, 10 ], [ 0, 0 ], 'k-')
plot([ 0, 0 ], [ -10, 10 ], 'k-')
hold off

xlabel('$\mathrm{Re}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$\mathrm{Im}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
title('Region of absolute stability of the Euler method', ...
    'FontSize', 16, 'Interpreter', 'latex')
axis([ -3, 1, -1.5, 1.5 ])
colormap([ 153, 204, 255 ; 255, 255, 255 ] / 255)
```



The Euler method for solving $y' = \lambda y$ will be stable for point z that lies within the shaded region.

Interval of absolute stability

The choice of step length used in a method will depend on accuracy requirements, computational resources available and of course stability. It is often necessary to use as large a value of the step length as possible permitted by the stability requirements to minimise the computational effort required to solve an ODE. The range values of the step length that can be chosen is governed by the stability region and provides use with the following definition.

Definition (interval of absolute stability)

The range of real values that the step length h of a method can take that ensures a method remains absolutely stable is known as the **interval of absolute stability**

The region of absolute stability for the Euler method plotted above shows that the interval of absolute stability is

$$z \in [-2, 0],$$

i.e., the real part of the region of absolute stability.

Since $z = h\lambda$ then

$$h \in \left[-\frac{2}{\lambda}, 0\right],$$

so we have the condition

$$h \leq -\frac{2}{\lambda}.$$

Example 4

The step length for the Euler method when used to solve the ODE $y' = -2.3y$ must satisfy

$$h \leq \frac{2}{2.3} \approx 0.8696.$$

This is why in example 1 the solution using $h = 0.7$ was stable since $0.7 < 0.8696$ and the solution using $h = 1.0$ was unstable since $1 > 0.8696$.

Stability function of a Runge-Kutta method

The general form of a Runge-Kutta method is

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i,$$

$$k_i = f(t_n + c_i h, y_n + h\sum_{j=1}^{s} a_{ij} k_k).$$

Let $Y_i = y_n + h\sum_{j=1}^{s} a_{ij}k_j$ and applying the method to the test ODE $y' = \lambda y$ the method becomes

$$y_{n+1} = y_n + h\lambda \sum_{i=1}^{s} b_i Y_i, \qquad (1)$$

$$Y_i = y_n + h\lambda \sum_{j=1}^{s} a_{ij} Y_j.$$

Let $z = h\lambda$ and expanding out the summations in the stage values gives

$$\begin{aligned} Y_1 &= y_n + z(a_{11}Y_1 + a_{12}Y_2 + \cdots + a_{1s}Y_s), \\ Y_2 &= y_n + z(a_{21}Y_1 + a_{22}Y_2 + \cdots + a_{2s}Y_s), \\ &\vdots \\ Y_s &= y_n + z(a_{s1}Y_1 + a_{s2}Y_2 + \cdots + a_{ss}Y_s). \end{aligned}$$

Let $Y = (Y_1, Y_2, \ldots, Y_s)^T$ and $\mathbf{e} = (1, 1, \ldots, 1)^T$ then we can write the stage values in vector form as

$$Y = \mathbf{e}y_n + zAY. \qquad (2)$$

Substituting $\sum_{i=1}^{s} b_i Y_i = \mathbf{b}^T \cdot Y$ into equation (1) gives the vector form of a Runge-Kutta method for solving the test ODE

$$y_{n+1} = y_n + z\mathbf{b}^T Y. \qquad (3)$$

Stability function of an explicit Runge-Kutta method

Rearranging equation (2) we have

$$Y = (I - zA)^{-1}\mathbf{e}y_n,$$

and substituting into equation (3) gives

$$\begin{aligned} y_{n+1} &= y_n + z\mathbf{b}^T(I - zA)^{-1} \cdot \mathbf{e}y_n \\ &= (1 + z\mathbf{b}^T(I - zA)^{-1}\mathbf{e})y_n, \end{aligned}$$

so the stability function is

$$R(z) = 1 + z\mathbf{b}^T(I - zA)^{-1}\mathbf{e}.$$

Using the geometric series of matrices

$$(I - zA)^{-1} = \sum_{k=0}^{\infty} (zA)^k,$$

the stability function can be written as the infinite series

$$R(z) = 1 + \left(\mathbf{b}^T \sum_{k=0}^{\infty} A^k \mathbf{e}\right) z^{k+1}. \qquad (4)$$

Since the solution to the test ODE is $y = e^{\lambda t}$, over one step of an Explicit Runge-Kutta (ERK) method we would expect the local truncation errors to change at a rate of e^z. The series expansion of e^z is

$$e^z = \sum_{k=0}^{\infty} \frac{1}{k!} z^k = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4 + \cdots \qquad (5)$$

Comparing the coefficients of z^k in equations (4) and (5) we have

$$\frac{1}{k!} = \mathbf{b}^T A^{k-1} \mathbf{e},$$

which must be satisfied up to the kth term for an order k ERK method to be stable.

Example 5

Determine the stability function for the following Runge-Kutta method and hence find its order.

$$\begin{array}{cccccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{3}{4} & 0 & \frac{3}{4} & & \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

The code below calculates the coefficients and outputs the stability function $R(z)$ for this ERK method.

```
clear

% Define ERK method
A = [ 0, 0, 0, 0 ;
      1/2, 0, 0, 0 ;
      0, 3/4, 0, 0 ;
      2/9, 1/3, 4/9, 0 ];
b = [ 7/24 ; 1/4 ; 1/3 ; 1/8 ];
c = [ 0 ; 1/2 ; 3/4 ; 1 ];
s = length(b);
e = ones(s, 1);

% Determine stability function
R = 'R(z) = 1';
for k = 1 : s
    coeff = b' * A^(k-1) * e;
    R = [ R, sprintf(' + %s z^%1i', strtrim(rats(coeff)), k) ];
end
fprintf(R)
```

```
R(z) = 1 + 1 z^1 + 1/2 z^2 + 3/16 z^3 + 1/48 z^4
```

So the stability function is

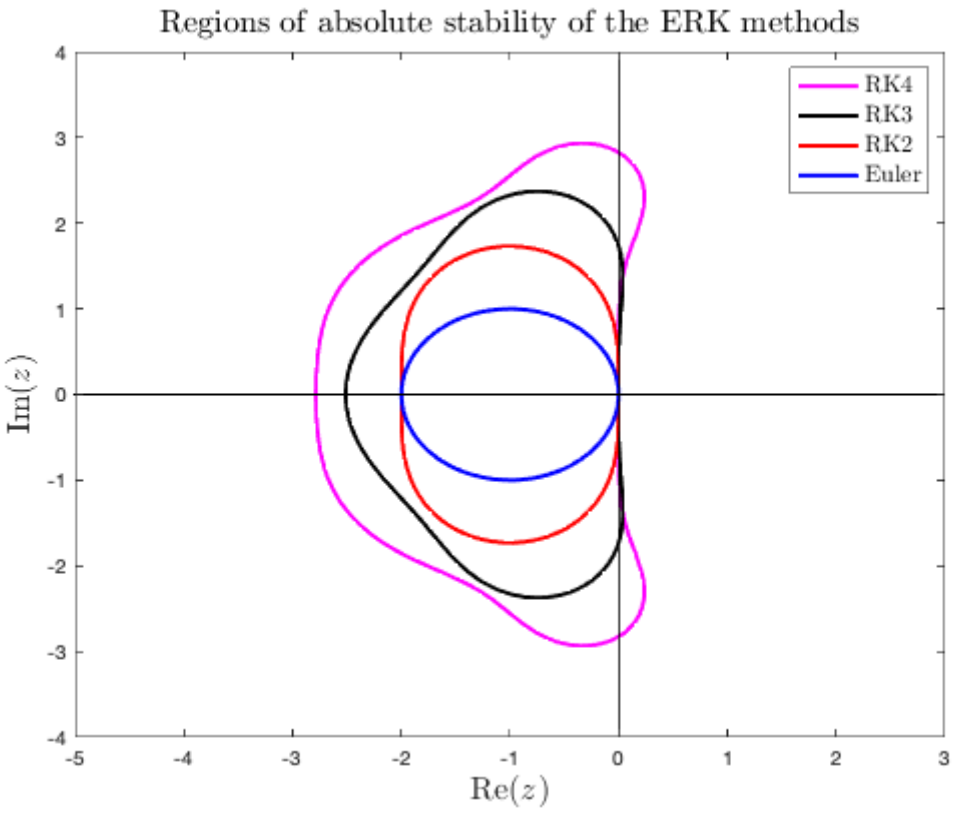
$$R(z) = 1 + z + \frac{1}{2}z^2 + \frac{3}{16}z^3 + \frac{1}{48}z^4,$$

which agrees to the series expansion of e^z up to and including the z^2 term. Therefore this ERK method is of order 2.

Example 6

The code below calculates the stability functions of order 1, 2, 3 and 4 ERK methods and plots their regions of absolute stability.

```
clear

% Generate z values
x = linspace(-5, 5, 100);
y = linspace(-5, 5, 100);
[X, Y] = meshgrid(x, y);
Z = X + Y * 1i;

% Define the stability functions for the ERK methods
R1 = 1 + Z;
R2 = R1 + 1/2 * Z.^2;
R3 = R2 + Z.^3 / 6;
R4 = R3 + Z.^4 / 24;

% Plot the region of absolute stability
contour(X, Y, abs(R4), [1, 1], 'm-', 'linewidth', 2)
hold on
contour(X, Y, abs(R3), [1, 1], 'k-', 'linewidth', 2)
contour(X, Y, abs(R2), [1, 1], 'r-', 'linewidth', 2)
contour(X, Y, abs(R1), [1, 1], 'b-', 'linewidth', 2)
plot([ -10, 10 ], [ 0, 0 ], 'k-')
plot([ 0, 0 ], [ -10, 10 ], 'k-')
hold off

axis([-5, 3, -4, 4])
xlabel('$\mathrm{Re}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$\mathrm{Im}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
title('Regions of absolute stability of the ERK methods', ...
    'FontSize', 16, 'Interpreter', 'latex')
leg = legend('RK4', 'RK3', 'RK2', 'Euler');
set(legend, 'Fontsize', 12, 'Interpreter', 'latex')
```



Stability functions of implicit methods

The simplest implicit method for solving ODEs is the Backwards Euler method (also known as the implicit Euler method) which is

$$y_{n+1} = y_n + hf(t_n + h, y_{n+1}).$$

Applying this to solve the test ODE $y' = \lambda y$ and rearranging gives

$$\begin{aligned} y_{n+1} &= y_n + h\lambda y_{n+1} \\ (1 - h\lambda)y_{n+1} &= y_n \\ y_{n+1} &= \frac{1}{1 - h\lambda} y_n, \end{aligned}$$

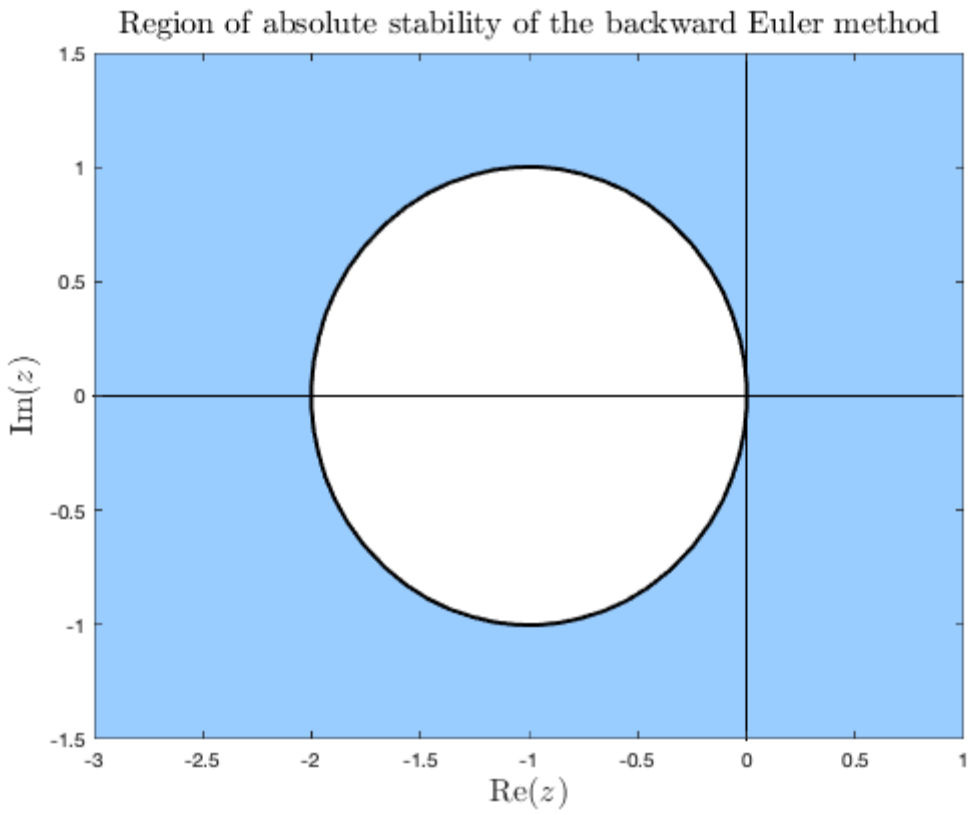
therefore the stability function for the backwards Euler method is $R(z) = \frac{1}{1-z}$. Stability functions for implicit methods take the form of a rational fraction

$$R(z) = \frac{P(z)}{Q(z)}.$$

Example 7

The code below plots the region of absolute stability of the backwards Euler method.

```
clear

% Generate z values
x = linspace(-5, 5, 100);
y = linspace(-5, 5, 100);
[X, Y] = meshgrid(x, y);
Z = X + Y * 1i;

% Define the stability function for the backwards Euler method
R = 1 ./ (1 + Z);

% Plot the region of absolute stability
contourf(X, Y, abs(R), [0 1], 'linewidth', 2)
hold on
plot([ -10, 10 ], [ 0, 0 ], 'k-')
plot([ 0, 0 ], [ -10, 10 ], 'k-')
hold off

xlabel('$\mathrm{Re}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$\mathrm{Im}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
title('Region of absolute stability of the backward Euler method', ...
    'FontSize', 16, 'Interpreter', 'latex')
axis([ -3, 1, -1.5, 1.5 ])
colormap([ 153, 204, 255 ; 255, 255, 255 ] / 255)
```



Here the region of absolute stability includes all of the complex plane with the exception of the unshaded region shown here.

Stability functions for an implicit Runge-Kutta method

To determine the stability function for an Implicit Runge-Kutta (IRK) method we use equations (3) and (2)

$$\begin{aligned} y_{n+1} &= y_n + z\mathbf{b}^T Y, \\ Y &= \mathbf{e}y_n + zAY. \end{aligned}$$

Transposing these equations so that the terms not involving y_n to the left-hand side gives

$$\begin{aligned} y_{n+1} - z\mathbf{b}^T Y &= y_n \\ (I - zA)Y &= \mathbf{e}y_n, \end{aligned}$$

which can be written as the matrix equation

$$\begin{pmatrix} 1 & -zb_1 & -zb_2 & \cdots & -zb_s \\ 0 & 1-za_{11} & -za_{12} & \cdots & -za_{1s} \\ 0 & -za_{21} & 1-za_{22} & \cdots & -za_{2s} \\ 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & -za_{s1} & -za_{s2} & \cdots & 1-za_{ss} \end{pmatrix} \begin{pmatrix} y_{n+1} \\ Y_1 \\ Y_2 \\ \vdots \\ Y_s \end{pmatrix} = \begin{pmatrix} y_n \\ y_n \\ \vdots \\ y_n \\ y_n \end{pmatrix}.$$

Using Cramer's rule to solve this system for y_{n+1} we have

$$y_{n+1} = \frac{\det \begin{pmatrix} y_n & -zb_1 & -zb_2 & \cdots & -zb_s \\ y_n & 1-za_{11} & -za_{12} & \cdots & -za_{1s} \\ y_n & -za_{21} & 1-za_{22} & \cdots & -za_{2s} \\ y_n & \vdots & \vdots & \ddots & \vdots \\ y_m & -za_{s1} & -za_{s2} & \cdots & 1-za_{ss} \end{pmatrix}}{\det(I - zA)}.$$

Performing a row operation of subtracting the first row of matrix in the numerator from the other rows gives

$$y_{n+1} = \frac{\det \begin{pmatrix} y_n & -zb_1 & -zb_2 & \cdots & -zb_s \\ 0 & 1-za_{11}+zb_1 & -za_{12}+zb_2 & \cdots & -za_{1s}+zb_s \\ 0 & -za_{21}+zb_1 & 1-za_{22}+zb_2 & \cdots & -za_{2s}+zb_s \\ 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & -za_{s1}+zb_1 & -za_{s2}+zb_2 & \cdots & 1-za_{ss}+zb_s \end{pmatrix}}{\det(I - zA)}$$

where $\mathbf{e}\mathbf{b}^T$ is a diagonal matrix with the elements of $\mathbf{b}$ on the main diagonal. Therefore the stability function of an Implicit Runge-Kutta (IRK) method can be written as

$$R(z) = \frac{\det(I - z(A - \mathbf{e}\mathbf{b}^T))}{\det(I - zA)}. \qquad (6)$$

A-stability

As we saw in the plot of the region of absolute stability of the backwards Euler method, implicit methods have a much greater stability region than explicit methods and are very useful for solving stiff ODEs where the stability constraints placed on an explicit method means the step length h is too small to be of practical use. A desirable property of some implicit methods is that there is no limit placed on the value of h for which will result in an unstable method, this is known as .

Definition (A-stability)

A method is said to be **A-stable** if its region of absolute stability satisfies

$$\{z : z \in \mathbb{C}^-, R(z) \leq 1\}$$

i.e., the method is stable for all points in the left-hand side of the complex plane.

Theorem

Given an implicit Runge-Kutta method with a stability function of the form

$$R(z) = \frac{P(z)}{Q(z)},$$

and define a polynomial function

$$E(y) = Q(iy)Q(-iy) - P(iy)P(-iy),$$

then the method is A-stable if and only if the following are satisfied

1. All roots of $Q(z)$ have positive real parts;
2. $E(y) \geq 0$ for all $y \in \mathbb{R}$.

Example 8

Determine the stability function of the following IRK and determine whether it is A-stable or not

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

```
clear

% Define ERK method
A = [ 5/12 , -1/12
      3/4, 1/4 ];
b = [ 3/4 , 0 ;
      0 , 1/4 ];
I = eye(size(b));
```

```
% Calculate P(z) and Q(z) polynomials
syms z
P = det(I - z * (A - b));
Q = det(I - z * A);

% Calculate roots of Q
Qroot = solve(Q == 0);

% Calculate E(y)
syms y
Pp = det(I + 1i * y * (A - b));
Pm = det(I - 1i * y * (A - b));
Qp = det(I + 1i * y * A);
Qm = det(I - 1i * y * A);
E = expand(Qp * Qm - Pp * Pm);

% Output P(z), Q(z), roots of Q(z) and E(y)
for i = 1 : 1
    fprintf('P(z) = %s\nQ(z) = %s', P, Q)
    fprintf('The roots of Q are %s and %s', Qroot(1), Qroot(2))
    fprintf('E(y) = %s', E)
end
```

```
P(z) = z/3 + z^2/16 + 1
Q(z) = z^2/6 - (2*z)/3 + 1
The roots of Q are 2 - 2^(1/2)*1i and 2^(1/2)*1i + 2
E(y) = y^2/8 + (55*y^4)/2304
```

So the stability function for this IRK method is

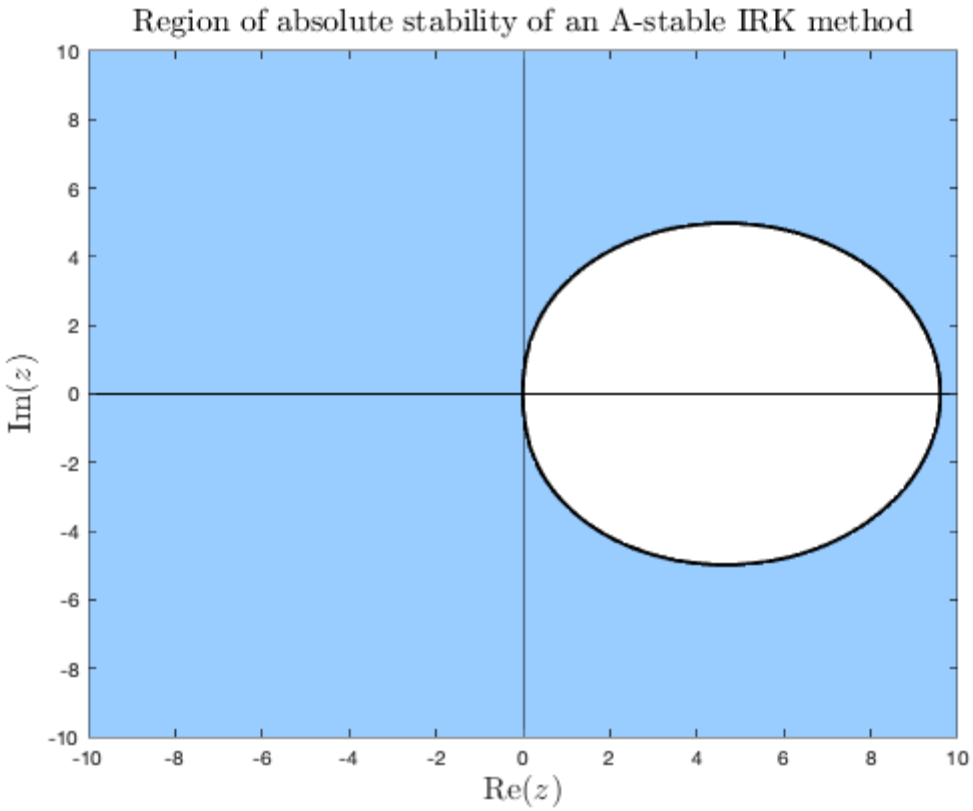
$$R(z) = \frac{1 + \frac{1}{3}z + \frac{1}{16}z^2}{1 - \frac{2}{3}z + \frac{1}{6}z^2}.$$

Since the real parts of both roots of $Q(z)$ is 2 which is positive and $E(y) = \frac{55}{2304}y^4 + \frac{1}{8}y^2 \geq 0$ for all $y \in \mathbb{R}$ we can say that this is an A-stable method.

The code below plots the region of absolute stability for this IRK method.

```
clear

% Generate z values
x = linspace(-10, 10, 100);
y = linspace(-10, 10, 100);
[X, Y] = meshgrid(x, y);
Z = X + Y * 1i;

% Define the stability function for the backwards Euler method
R = (1 + 1/3 * Z + 1/16 * Z .^ 2) ./ (1 - 2/3 * Z + 1/6 * Z.^2);

% Plot the region of absolute stability
contourf(X, Y, abs(R), [0 1], 'linewidth', 2)
hold on
plot([ -10, 10 ], [ 0, 0 ], 'k-')
plot([ 0, 0 ], [ -10, 10 ], 'k-')
hold off

xlabel('$\mathrm{Re}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
ylabel('$\mathrm{Im}(z)$', 'FontSize', 16, 'Interpreter', 'latex')
title('Region of absolute stability of an A-stable IRK method', ...
```

```
    'FontSize', 16, 'Interpreter', 'latex')
axis([ -10, 10, -10, 10 ])
colormap([ 153, 204, 255 ; 255, 255, 255 ] / 255)
```



Stiffness

An important consideration when applying numerical methods to solve ODEs is whether the values that the step length can take for the method to be stable are large enough for the method to be applied without incurring prohibitive computational costs. For example, if a method requires a very small step length for it to be stable, then the iterations required to step through the domain will be very large resulting in lots of computational operations and therefore time. Problems like is a known as stiff problems.

Definition (stiffness)

If a numerical method is forced to use, in a certain interval of integration, a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the problem is said to be **stiff** in that interval (Lambert, 1990).

The stiffness of a problem depends on the system ODEs and the solver being applied to solve them and stiffness usually arises when there is a large variation in the behaviour of the individual ODEs in the system. Stiff systems require methods which are stable for larger values of the step length h, e.g., implicit or A-stable methods.

Stiffness ratio

Consider a system of linear ODEs of the form

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

where A is a coefficient matrix. We use the test ODE $y' = \lambda y$ to examine the stability of a numerical method so we need to transform our system into this form. Let u_i be a transformation of y_i and $u_i' = \lambda_i u_i$ where λ_i is an eigenvalue of A then

$$\begin{pmatrix} u_1' \\ u_2' \\ \vdots \\ u_N' \end{pmatrix} = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_N \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix}.$$

Let $V = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N)$ be a matrix containing the eigenvectors $\mathbf{v}_i$ corresponding to the eigenvalues λ_i of A then

$$AV = (A\mathbf{v}_1, A\mathbf{v}_2, \ldots, A\mathbf{v}_N),$$

and since $\lambda_i = A\mathbf{v}_i$

$$\begin{aligned} AV &= (\lambda_1\mathbf{v}_1, \lambda_2\mathbf{v}_2, \ldots, \lambda_N\mathbf{v}_N) = V\Lambda \\ V^{-1}AV &= \Lambda. \end{aligned}$$

So $\Lambda = V^{-1}AV$ is a linear transformation that diagonalises A and the stability behaviour of $\mathbf{y}' = A\mathbf{y}$ can be analysed by considering $\mathbf{y}' = \Lambda\mathbf{y}$. Since a stiff system will have a large variation in the step length and hence the values of λ_i we can define a **stiffness ratio**

$$S = \frac{\max |\mathrm{Re}(\lambda_i)|}{\min |\mathrm{Re}(\lambda_i)|}.$$

Example 9

Determine the stiffness ratio of the following ODE

$$y'' - 1001y' - 1000y = 0.$$

Writing this as a system of first-order ODEs

$$\begin{aligned} y_1' &= y_2', \\ y_2' &= 1001y_2 - 1000y_1, \end{aligned}$$

which can be written as the matrix equation

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1000 & 1001 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}.$$

The eigenvalues of the coefficient matrix are $\lambda_1 = 1$ and $\lambda_2 = 1000$ so

$$S = \frac{1000}{1} = 1000,$$

and this ODE can be considered stiff.

Summary

- The local truncation error is the error at each step of a method due to the truncation of the Taylor series in deriving the method.
- The global truncation error is the accumulation of the local truncation errors.
- A method is considered absolutely stable if the local truncation errors do not increase from one step to the next.
- The stability of a method is analysed by considering the simple test ODE $y' = \lambda y$.
- The stability function of a method is the amount by which the solution to the test ODE changes over a single step of length.
- The region of absolute stability is the set of all values in the complex plane where the modulus of the stability function is less than or equal to 1 (i.e., the local truncation errors do not increase over a single step).
- The stability function for an explicit Runge-Kutta method is given in equation (4).
- The stability function for an order k explicit Runge-Kutta method is the kth-order series expansion of e^z given in equation (5).
- The stability function for an implicit Runge-Kutta method is given in equation (6).
- Implicit methods can be A-stable which means there is no limit placed on the value of the step length h for a method to remain stable.
- A system of ODEs is considered stiff if the ratio of the eigenvalues of the coefficient matrix is large. Stiff systems require methods that have larger regions of absolute stability.

Exercises

1. Determine the stability function of the following Runge-Kutta method

$$\begin{array}{ccccccc} 0 & & & & & & \\ \frac{1}{4} & \frac{1}{4} & & & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & & & \\ \frac{3}{4} & \frac{1}{2} & \frac{1}{4} & & & & \\ 0 & 0 & \frac{1}{6} & -\frac{1}{3} & \frac{1}{6} & & \\ & -1 & \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} & 1 & \end{array}$$

2. Determine the stability function of the following Runge-Kutta method. Is this an A-stable method?

$$\begin{array}{ccccc} \frac{1}{4} & \frac{7}{24} & -\frac{1}{24} \\ \frac{3}{4} & \frac{13}{24} & \frac{5}{24} \\ & \frac{1}{2} & \frac{1}{2} \end{array}$$

3. Plot the region of absolute stability for the following Runge-Kutta method.

$$\begin{array}{ccc} \frac{1}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 0 \\ & \frac{3}{4} & \frac{1}{4} \end{array}$$

4. Determine the stability function of the following Runge-Kutta method. What is the order of the method?

$$\begin{array}{ccccccc} 0 & & & & & & \\ \frac{2}{7} & \frac{2}{7} & & & & & \\ \frac{4}{7} & -\frac{8}{35} & \frac{4}{5} & & & & \\ \frac{6}{7} & \frac{29}{42} & -\frac{2}{3} & \frac{5}{6} & & & \\ 1 & \frac{1}{6} & \frac{1}{6} & \frac{5}{12} & \frac{1}{4} & & \\ & \frac{11}{96} & \frac{7}{24} & \frac{35}{96} & \frac{7}{48} & \frac{1}{12} & \end{array}$$

5. Calculate the stiffness ratio for the following system of ODEs.

$$y_1' = -80.6y_1 + 119.4y_2,$$
$$y_2' = 79.6y_1 - 120.4y_2.$$

What are the maximum step lengths that the Euler method is stable for solving each equation?

References

Lambert, J.D. (1991) *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem.* New York, NY, USA: John Wiley & Sons, Inc.

Direct Methods for Solving Systems of Linear Equations

Learning outcomes

On successful completion of this chapter readers will be able to:

- Apply LU decomposition to factorise a square matrix into a product of a lower triangular and upper triangular matrices.
- Apply Crout's method for solving a system of linear equations using LU decomposition.
- Solve a system of linear equations using LU decomposition with partial pivoting.
- Apply Cholesky decomposition to factorise a positive definite matrix into a product of a lower triangular matrix and its transpose.
- Solve a system of linear equations using the Cholesky-Crout method.
- Apply QR decomposition to factorise an $m \times n$ matrix into the product of an orthogonal matrix and an upper triangular matrix.
- Solve a systems of linear equations using QR decomposition.

Systems of linear equations

Linear systems of equations appear often in the topics of numerical analysis and numerical solutions to differential equations. The methods that are applied to solve systems of linear equations fall into one of two categories: **direct methods** that use an algebraic approach and **indirect methods** that use an iterative approach. On this page we will look at some common direct methods.

A **system of linear equations** with m equations and n unknowns can be expressed as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_n, \end{aligned}$$

where x_i are the unknowns, a_{ij} are coefficients and b_i are constant terms. It is often more convenient to express a system of linear equations as a matrix equation. Let $[A]_{ij} = a_{ij}$ be the **coefficient matrix**, $\mathbf{x} = (x_1, x_2, \ldots, x_m)^T$ be the **unknown vector** and $\mathbf{b} = (b_1, b_2, \ldots, b_m)^T$ be the **constant vector** then we can rewrite a system of linear equations as $A\mathbf{x} = \mathbf{b}$, i.e.,

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

LU decomposition

LU decomposition (also known as **LU factorisation**) is a procedure for factorising a square matrix A into the product of a **lower triangular** matrix L and an **upper triangular** matrix U such that

$$A = LU.$$

The advantage of writing a matrix as a product of L and U is that the solution to a triangular set of equations is easy to calculate using forward and back substitution.

Consider the LU decomposition of a 3×3 matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix},$$

which gives a system of 9 equations (one for each element in A) in 12 unknowns which has an infinite number of solutions. If we use the condition $\ell_{ii} = 1$ then

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ \ell_{21}u_{11} & \ell_{21}u_{12} + u_{22} & \ell_{21}u_{13} + \ell_{22}u_{23} \\ \ell_{31}u_{11} & \ell_{31}u_{12} + \ell_{32}u_{22} & \ell_{31}u_{13} + \ell_{32}u_{23} + \ell_{33}u_{33} \end{pmatrix}$$

The elements in the lower triangular region $(i > j)$ are

$$a_{ij} = \sum_{k=1}^{j} \ell_{ik}u_{kj} = \ell_{ij}u_{jj} + \sum_{k=1}^{j-1} \ell_{ik}u_{kj},$$

which is rearranged to

$$\ell_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}u_{kj} \right). \qquad (1)$$

For the elements in the upper triangular region $(i \leq j)$ we have

$$a_{ij} = u_{ij} + \sum_{k=1}^{i-1} \ell_{ik}u_{kj},$$

which is rearranged to

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj}. \qquad (2)$$

So to calculate the LU decomposition of a square matrix A we loop through each column of A and calculate the elements of L and U for that column using equations (1) and (2), i.e.,

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj}, \qquad i = 1, 2, \ldots, j, \qquad (3)$$

$$\ell_{ij} = \begin{cases} 1, & i = j, \\ \dfrac{1}{u_{jj}} \left(a_{ij} - \displaystyle\sum_{k=1}^{j-1} \ell_{ik}u_{kj} \right), & i = j+1, j+2, \ldots, n \end{cases} \qquad (4)$$

Example 1

Determine the LU decomposition of the following matrix

$$A = \begin{pmatrix} 1 & 3 & 0 \\ 2 & -4 & -1 \\ -3 & 1 & 2 \end{pmatrix}.$$

Stepping through the columns of A

$$\begin{aligned}
j = 1: \quad & u_{11} = a_{11} = 1, \\
& \ell_{21} = \frac{1}{u_{11}}(a_{21}) = \frac{1}{1}(2) = 2, \\
& \ell_{31} = \frac{1}{u_{11}}(a_{31}) = \frac{1}{1}(-3) = -3, \\
j = 2: \quad & u_{12} = a_{12} = 3, \\
& u_{22} = a_{22} - \ell_{21}u_{12} = -4 - 2(3) = -10, \\
& \ell_{32} = \frac{1}{u_{22}}(a_{32} - \ell_{31}u_{12}) = \frac{1}{-10}(1 + 3(3)) = 1, \\
j = 3: \quad & u_{13} = a_{13} = 0, \\
& u_{23} = a_{23} - \ell_{21}u_{13} = -1 - 2(0) = -1, \\
& u_{33} = a_{33} - \ell_{31}u_{13} - \ell_{32}u_{23} = 2 + -3(0) - 1(-1) = 3.
\end{aligned}$$

Therefore

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 1 & 3 & 0 \\ 0 & -10 & -1 \\ 0 & 0 & 1 \end{pmatrix}.$$

Checking that $LU = A$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 3 & 0 \\ 0 & -10 & -1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 0 \\ 2 & -4 & -1 \\ -3 & 1 & 2 \end{pmatrix}.$$

Example 2

The function called LUdecomp below calculates the LU decomposition of a square matrix `A`.

```
function [L, U] = LUdecomp(A)

% Calculates the LU decomposition of the square matrix A

% Initialise L and U
ncols = size(A, 1);
L = eye(ncols);
U = zeros(ncols);

% Loop through columns
```

```
    for j = 1 : ncols

        % Calculate u_ij for i < = j
        for i = 1 : j
            for k = 1 : i - 1
                U(i, j) = U(i, j) + L(i, k) * U(k, j);
            end
            U(i, j) = A(i, j) - U(i, j);
        end

        % Calculate l_ij for i > j
        for i = j + 1 : ncols
            for k = 1 : j - 1
                L(i, j) = L(i, j) + L(i, k) * U(k, j);
            end
            L(i, j) = (A(i, j) - L(i, j)) / U(j, j);
        end
    end

    end
```

The code below uses the function `LUdecomp` to calculate the LU decomposition of the matrix from example 1.

```
% Define matrix A
A = [ 1, 3, 0 ;
      2, -4, -1 ;
     -3, 1, 2 ];

% Calculate L and U
[L, U] = LUdecomp(A)
```

```
L = 3×3
     1     0     0
     2     1     0
    -3    -1     1
U = 3×3
     1     3     0
     0   -10    -1
     0     0     1
```

```
% Check that A - L.U = 0
A - L * U
```

```
ans = 3×3
     0     0     0
     0     0     0
     0     0     0
```

Crout's method

Given a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ then the solution can be calculated using the LU decomposition of A using **Crout's method**. Since $A = LU$ then

$$LU\mathbf{x} = \mathbf{b}$$

Let $\mathbf{y} = U\mathbf{x}$ then

$$L\mathbf{y} = \mathbf{b}.$$

L is lower triangular so the solution for $L\mathbf{y} = \mathbf{b}$ is easily calculated using forward substitution. Once $\mathbf{y}$ has been calculated the solution to $U\mathbf{x} = \mathbf{y}$ is calculated using back substitution.

The advantage of using Crout's method is that once the LU decomposition of the coefficient matrix has been calculated the can be used for any values of the right-hand side vector $\mathbf{b}$ unlike Gaussian elimination where row reduction will need to be repeated for difference values of $\mathbf{b}$.

Example 3

Use Crout's method to solve the following system of linear equations

$$\begin{pmatrix} 1 & 3 & 0 \\ 2 & -4 & -1 \\ -3 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -7 \\ 11 \\ 1 \end{pmatrix}.$$

We saw in example 1 that the LU decomposition of the coefficient matrix is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 1 & 3 & 0 \\ 0 & -10 & -1 \\ 0 & 0 & 1 \end{pmatrix}.$$

Solving $L\mathbf{y} = \mathbf{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -7 \\ 11 \\ 1 \end{pmatrix},$$

gives

$$\begin{aligned} y_1 &= -7, \\ y_2 &= 11 - 2y_1 = -2(-7) = 25, \\ y_3 &= -1 + 3y_1 + y_2 = -1 + 3(-7) + 1(25) = 5. \end{aligned}$$

Solving $U\mathbf{x} = \mathbf{y}$

$$\begin{pmatrix} 1 & 3 & 0 \\ 0 & -10 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -7 \\ 25 \\ 5 \end{pmatrix},$$

gives

$$\begin{aligned} x_3 &= \frac{1}{1} x_3 = 5, \\ x_2 &= \frac{1}{-10}(25 + x_3) = -\frac{1}{10}(25 + 5) = -3, \\ x_1 &= \frac{1}{1}(-7 - 0x_3 - 3x_2) = -7 + 9 = 2. \end{aligned}$$

So the solution is $\mathbf{x} = (2, -3, 5)$.

Example 4

The function called `crout` below calculates the solution to the system of linear equations $LU\mathbf{x} = \mathbf{b}$ using Crout's method.

```
function x = crout(L, U, b)

% Calculates the solution to the system of linear equations LUx=b using
% Crouts method

% Solve Ly = b using forward substitution
y = forward_sub(L, b);

% Solve Ux = y using back substitution
x = back_sub(U, y);

end
```

The function `crout` uses the functions `forward_sub` and `back_sub` shown below which perform forward and back substitution.

```
function x = forward_sub(L, b)

% Calculates the solution to Lx = b where L is a lower triangular matrix
% using forward substitution

ncols = length(b);
x = zeros(ncols, 1);
for i = 1 : ncols
    for j = 1 : i - 1
        x(i) = x(i) + L(i, j) * x(j);
    end
    x(i) = (b(i) - x(i)) / L(i, i);
end

end
```

```
function x = back_sub(L, b)

% Calculates the solution to Ux = b where U is an upper triangular matrix
% using back substitution

ncols = length(b);
x = zeros(ncols, 1);
for i = ncols : -1 : 1
    for j = i : ncols
        x(i) = x(i) + U(i, j) * x(j);
    end
    x(i) = (b(i) - x(i)) / U(i, i);
end
```

```
end
```

The code below uses the function `crout` to solve the system of linear equations from example 3.

```
clear

% Define system of linear equations
A = [ 1, 3, 0 ;
      2, -4, -1 ;
     -3, 1, 2 ];
b = [ -7 ; 11 ; 1 ];

% Calculate LU decomposition of A
[L, U] = LUdecomp(A);

% Calculate solution to Ax = b using Crout's method
x = crout(L, U, b);

% Output results
for i = 1 : length(x)
    fprintf('x_%1i = %1.4f\n', i, x(i))
end
```

```
x_1 = 2.0000
x_2 = -3.0000
x_3 = 5.0000
```

Partial pivoting

A problem that can be encountered with LU decomposition is that if the value of u_{jj} in equation (1) is zero or some small number it will mean that ℓ_{ij} is undefined or prone to computational rounding errors due to the resulting value being very large (this is known as an ill-conditioned system).

This problem can be overcome by using **partial pivoting** where rows of the coefficient matrix are permuted so that the pivot element on the main diagonal is the larger than the elements in the column beneath it. The permutations applied to the coefficient matrix are recorded in a matrix P which is determined by applying the same permutations to the identity matrix.

Example 5

Apply partial pivoting the following matrix and determine the permutation matrix P

$$A = \begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 3 & -2 & 2 \end{pmatrix}.$$

Using row operations

$$\begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 3 & -2 & 2 \end{pmatrix} R_1 \leftrightarrow R_3 \quad \longrightarrow \quad \begin{pmatrix} 3 & -2 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & -2 \end{pmatrix} R_2 \leftrightarrow R_3 \quad \longrightarrow \quad \begin{pmatrix} 3 & -2 & 2 \\ 0 & 1 & -2 \\ 1 & 0 & 2 \end{pmatrix}.$$

Apply the same row operations to the identity matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} R_1 \leftrightarrow R_3 \quad \longrightarrow \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} R_2 \leftrightarrow R_3 \quad \longrightarrow \quad \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

So $P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. Note that PA gives the matrix A after partial pivoting has been applied i.e.,

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 3 & -2 & 2 \end{pmatrix} = \begin{pmatrix} 3 & -2 & 2 \\ 0 & 1 & -2 \\ 1 & 0 & 2 \end{pmatrix}.$$

Example 6

The function called `partial_pivot` below calculates the permutation matrix P for applying partial pivoting to the square matrix `A`.

```
function P = partial_pivot(A)

% Applies partial pivotting to A and returns the permutation matrix P

ncols = size(A, 1);
P = eye(ncols);

% Loop through columns
for j = 1 : ncols

    % Look for max pivot
    maxpivot = A(j, j);
    k = j;
    for i = j : ncols
        if abs(A(i,j)) > abs(maxpivot)
            maxpivot = A(i, j);
            k = i;
        end
    end

    % Swap pivot row with max pivot row
    temp = P(j, :);
    P(j, :) = P(k, :);
    P(k, :) = temp;
end

end
```

The code below uses the function `partial_pivot` to compute the permutation matrix P for the matrix A from example 5.

```
clear

% Define A matrix
A = [ 0, 1, -2 ;
```

```
      1, 0, 2 ;
      3, -2, 2 ];

% Calculate permutation matrix P
P = partial_pivot(A);
P
```

```
P = 3×3
     0     0     1
     1     0     0
     0     1     0
```

LU decomposition with partial pivoting

To calculate **LU decomposition with partial pivoting** uses the process as before with the exception that the coefficient matrix has partial pivoting applied prior to the calculation of L and U, i.e.,

$$LU = PA.$$

Using partial pivoting on the coefficient matrix A when solving the system of linear equations $A\mathbf{x} = \mathbf{b}$ results in

$$PA\mathbf{x} = \mathbf{b},$$

and since $P^{-1} = P$ (the inverse operation of swapping any two rows is to simply swap them back) then

$$A\mathbf{x} = P\mathbf{b}.$$

So Crout's method when using LU decomposition with partial pivoting requires solving the following for $\mathbf{y}$ and $\mathbf{x}$

$$L\mathbf{y} = P\mathbf{b},$$
$$U\mathbf{x} = \mathbf{y}.$$

Example 7

Solve the following system of linear equations using Crout's method with LUP decomposition

$$\begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 3 & -2 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ -4 \\ -8 \end{pmatrix}.$$

We have seen from example 5 that applying partial pivoting to the coefficient matrix results in

$$A = \begin{pmatrix} 3 & -2 & 2 \\ 0 & 1 & -2 \\ 1 & 0 & 2 \end{pmatrix}, \qquad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Calculating the LU decomposition of A using equations (3) and (4)

$$
\begin{aligned}
j=1: \quad & u_{11}=a_{11}=3, \\
& \ell_{21}=\frac{1}{u_{11}}a_{21}=\frac{1}{3}(0)=0, \\
& \ell_{31}=\frac{1}{u_{11}}a_{31}=\frac{1}{3}(1)=\frac{1}{3}, \\
j=2: \quad & u_{12}=a_{12}=-2, \\
& u_{22}=a_{22}-\ell_{21}u_{12}=1-0(-2)=1, \\
& \ell_{32}=\frac{1}{u_{22}}(a_{32}-\ell_{31}u_{12})=\frac{1}{1}\left(0-\frac{1}{3}(-2)\right)=\frac{2}{3}, \\
j=3: \quad & u_{13}=a_{13}=2, \\
& u_{23}=a_{23}-\ell_{21}u_{12}=-2-0(-2)=-2, \\
& u_{33}=a_{33}-\ell_{31}u_{13}-\ell_{32}u_{23}=2-\frac{1}{3}(2)-\frac{2}{3}(-2)=\frac{8}{3}.
\end{aligned}
$$

Therefore

$$
L=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{3} & \frac{2}{3} & 1 \end{pmatrix}, \qquad R=\begin{pmatrix} 3 & -2 & 2 \\ 0 & 1 & -2 \\ 0 & 0 & \frac{8}{3} \end{pmatrix}.
$$

Solving $L\mathbf{y}=P\mathbf{b}$ using forward substitution

$$
\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{3} & \frac{2}{3} & 1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}=\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 10 \\ -4 \\ -8 \end{pmatrix}=\begin{pmatrix} -8 \\ 10 \\ -4 \end{pmatrix},
$$

gives

$$
\begin{aligned}
& y_1=-8 \\
& y_2=10-0(8)=10, \\
& y_3=-4-\frac{1}{3}(8)+\frac{2}{3}(10)=-8.
\end{aligned}
$$

Solving $U\mathbf{x}=\mathbf{y}$ using back substitution

$$
\begin{pmatrix} 3 & -2 & 2 \\ 0 & 1 & -2 \\ 0 & 0 & \frac{8}{3} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}=\begin{pmatrix} -8 \\ 10 \\ -8 \end{pmatrix},
$$

gives

$$x_3 = \frac{3}{8}(-8) = -3,$$

$$x_2 = \frac{1}{1}(10 + 2(-3)) = 4,$$

$$x_1 = \frac{1}{3}(-8 + 2(4) - 2(-3)) = 2.$$

So the solution is $\mathbf{x} = (2, 4, -3)$.

Example 8

The code below uses the previously defined functions `partial_pivot`, `LUdecomp` and `crout` to solve the system of linear equations from example 7 using LU decomposition with partial pivoting.

```
clear

% Define system of linear equations
A = [ 0, 1, -2 ;
      1, 0, 2 ;
      3, -2, 2 ];
b = [ 10 ; -4 ; -8 ];

% Calculate permutation matrix P
P = partial_pivot(A);

% Calculate LU decomposition of PA
[L, U] = LUdecomp(P * A);

% Use Crout's method to solve LUx = Pb
x = crout(L, U, P * b);

% Output results
for i = 1 : length(x)
    fprintf('x_%1i = %1.4f\n', i, x(i))
end
```

```
x_1 = 2.0000
x_2 = 4.0000
x_3 = -3.0000
```

Cholesky decomposition

Cholesky decomposition is an efficient decomposition method that can be used when a square matrix is **positive definite**.

Definition (positive definite matrix)

A symmetric square matrix A is said to be positive definite if $\mathbf{x}^T A\mathbf{x}$ is positive for a non-zero column vector $\mathbf{x}$.

Given a positive definite matrix A then Cholesky decomposition factorises A into the product of a lower triangular matrix L and its transpose, i.e.,

$$A = LL^T.$$

Consider the Cholesky decomposition of a 3×3 matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{pmatrix} \begin{pmatrix} \ell_{11} & \ell_{21} & \ell_{31} \\ 0 & \ell_{22} & \ell_{32} \\ 0 & 0 & \ell_{33} \end{pmatrix} = \begin{pmatrix} \ell_{11}^2 & \ell_{11}\ell_{21} & \ell_{11}\ell_{31} \\ \ell_{11}\ell_{21} & \ell_{21}^2 + \ell_{22}^2 & \ell_{21}\ell_{31} + \ell_{22}\ell_{33} \\ \ell_{11}\ell_{31} & \ell_{21}\ell_{31} + \ell_{22}\ell_{33} & \ell_{31}^2 + \ell_{32}^2 + \ell_{33}^2 \end{pmatrix}.$$

The elements on the main diagonal are

$$a_{jj} = \ell_{jj}^2 + \sum_{k=1}^{j-1} \ell_{jk}^2,$$

and the other elements are

$$a_{ij} = \sum_{k=1}^{j} \ell_{ik}\ell_{jk} = \ell_{jj}\ell_{ij} + \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk}.$$

Rearranging these two expressions gives

$$\ell_{ij} = \begin{cases} \sqrt{a_{ii} - \sum_{k=1}^{j-1} \ell_{jk}^2}, & i = j, \\ \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{j=1}^{i-1} \ell_{ik}\ell_{jk} \right), & j = i+1, \ldots, n. \end{cases}$$

Example 9

Calculate the Cholesky decomposition of the following matrix

$$A = \begin{pmatrix} 4 & -2 & -4 \\ -2 & 10 & 5 \\ -4 & 5 & 14 \end{pmatrix}.$$

Steping through the columns of A

$$
\begin{aligned}
j=1: \quad & \ell_{11} = \sqrt{a_{11}} = \sqrt{4} = 2, \\
& \ell_{21} = \frac{1}{\ell_{11}}(a_{21}) = \frac{1}{2}(-2) = -1, \\
& \ell_{31} = \frac{1}{\ell_{11}}(a_{31}) = \frac{1}{2}(-4) = -2, \\
j=2: \quad & \ell_{22} = \sqrt{a_{22} - \ell_{21}^2} = \sqrt{10 - (-1)^2} = \sqrt{9} = 3, \\
& \ell_{32} = \frac{1}{\ell_{22}}(a_{32} - \ell_{31}\ell_{21}) = \frac{1}{3}(5 - (-2)(-1)) = 1, \\
j=3: \quad & \ell_{33} = \sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2} = \sqrt{14 - (-2)^2 - 1^2} = \sqrt{9} = 3,
\end{aligned}
$$

therefore $L = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ -2 & 1 & 3 \end{pmatrix}$. Checking that $A = LL^T$

$$
\begin{pmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ -2 & 1 & 3 \end{pmatrix}
\begin{pmatrix} 2 & -1 & -2 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix}
= \begin{pmatrix} 4 & -2 & -4 \\ -2 & 10 & 5 \\ -4 & 5 & 14 \end{pmatrix}.
$$

Example 10

The function called `cholesky` below calculates the Cholesky decomposition of the positive definite matrix `A`.

```
function L = cholesky(A)

% Calculates the Cholesky decomposition of a positive definite matrix A

ncols = size(A, 1);
L = zeros(ncols);

% Loop through columns
for j = 1 : ncols

    % Calculate main diagonal element
    for k = 1 : j - 1
        L(j, j) = L(j, j) + L(j, k) ^ 2;
    end
    L(j, j) = sqrt(A(j, j) - L(j, j));

    % Calculate lower triangular elements
    for i = j + 1 : ncols
        for k = 1 : j - 1
            L(i, j) = L(i, j) + L(i, k) * L(j, k);
        end
        L(i, j) = (A(i, j) - L(i, j)) / L(j, j);
    end
end

end
```

The code below uses the function `cholesky` to calculate the Cholesky decomposition of the matrix from example 9.

```
clear

% Define matrix A
A = [ 4, -2, -4 ;
     -2, 10, 5 ;
     -4, 5, 14 ];

% Calculate Cholesky decomposition
L = cholesky(A)
```

```
L = 3×3
     2     0     0
    -1     3     0
    -2     1     3
```

```
% Check that A - LL^T = 0
A - L * L'
```

```
ans = 3×3
     0     0     0
     0     0     0
     0     0     0
```

The Cholesky-Crout method

The **Cholesky-Crout method** is used to solve a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ where A is a positive definite matrix.

Let $\mathbf{y} = L^T\mathbf{x}$ then since $A = LL^T$ then

$$L\mathbf{y} = \mathbf{b},$$
$$L^T\mathbf{x} = \mathbf{y}.$$

Example 11

Solve the following system of linear equations using the Cholesky-Crout method.

$$\begin{pmatrix} 4 & -2 & -4 \\ -2 & 10 & 5 \\ -4 & 5 & 14 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -2 \\ 49 \\ 27 \end{pmatrix}.$$

We saw in example 9 that $L = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ -2 & 1 & 3 \end{pmatrix}$. Solving $L\mathbf{y} = \mathbf{b}$

$$\begin{pmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ -2 & 1 & 3 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -2 \\ 49 \\ 27 \end{pmatrix},$$

gives

$$y_1 = \frac{-2}{2} = -1,$$
$$y_2 = \frac{1}{3}(49 + y_1) = \frac{1}{3}(49 - 1) = 16,$$
$$y_3 = \frac{1}{3}(27 + 2y_1 - y_2) = \frac{1}{3}(27 + 2(-1) - 16) = 3.$$

Solving $L^T\mathbf{x} = \mathbf{y}$

$$\begin{pmatrix} 2 & -1 & -2 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ 16 \\ 3 \end{pmatrix},$$

gives

$$x_3 = \frac{3}{3} = 1,$$
$$x_2 = \frac{1}{3}(16 - x_3) = \frac{1}{3}(16 - 1) = 5,$$
$$x_1 = \frac{1}{2}(-1 + x_2 + 2x_3) = \frac{1}{2}(-1 + 5 + 2(1)) = 3.$$

So the solution is $\mathbf{x} = (3, 5, 1)$.

Example 12

The code below using the functions `cholesky` and `crout` to solve the system of linear equations from example 11 using the Cholesky-Crout method.

```
clear

% Define linear system
A = [ 4, -2, -4 ;
     -2, 10, 5 ;
     -4, 5, 14 ];
b = [ -2 ; 49 ; 27 ];

% Calculate Cholesky decomposition
L = cholesky(A);

% Solve linear system using the Cholesky-Crout method
x = crout(L, L', b);

% Output results
for i = 1 : length(x)
    fprintf('x_%1i = %1.4f\n', i, x(i))
end
```

```
x_1 = 3.0000
x_2 = 5.0000
x_3 = 1.0000
```

QR decomposition

The methods of LU and Cholesky decomposition can be used to solve a system of linear equations where the number of equations is the same as the number of unknowns so it has a single unique solution (if it exists). QR factorisation can be used to solve an **overdetermined** system where the number of equations exceeds the number of unknowns. Overdetermined systems rarely have a unique solution but we can calculate an approximation that most closely satisfies all equations in the system.

The $m \times n$ coefficient matrix A is factorised into the product of two matrices such that

$$A = QR,$$

where Q is an orthogonal matrix and R is an upper triangular matrix.

Definition (orthogonal vectors)

A set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \dots\}$ is said to be **orthogonal** if $\mathbf{v}_i \cdot \mathbf{v}_j = 0$ for $i \neq j$. Furthermore the set is said to be **orthonormal** if

$$\mathbf{v}_i \cdot \mathbf{v}_j = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Definition (orthogonal matrix)

An **orthogonal matrix** is a matrix where the columns are a set of orthonormal vectors. If A is an orthogonal matrix if

$$A^T A = I.$$

Example 13

Show that the following matrix is an orthogonal matrix

$$A = \begin{pmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{pmatrix}.$$

Checking $A^T A = I$

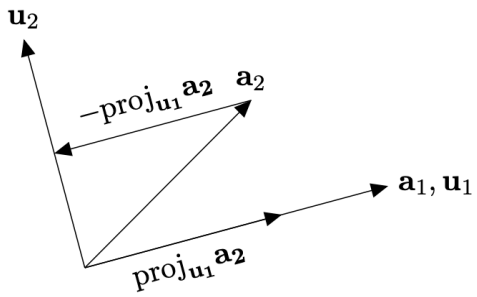
$$A^T A = \begin{pmatrix} 0.8 & 0.6 \\ -0.6 & 0.8 \end{pmatrix} \begin{pmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I.$$

So A is an orthogonal matrix.

Calculating QR decomposition using the Gram-Schmidt process

The calculation of the orthogonal matrix Q can be achieved by using the **Gram-Schmidt process**. Given a matrix consisting of n linearly independent column vectors $A = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)$ we wish to find

a matrix that consists of nn orthogonal vectors $U = (\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n)$ where the span of U is that same as the span of A.



Consider the diagram above. Let $\mathbf{u}_1 = \mathbf{a}_1$ then the vector $\mathbf{u}_2$ this is orthogonal to $\mathbf{u}_1$ can be found by subtracting the **vector projection** of $\mathbf{a}_2$ onto $\mathbf{u}_1$, i.e.,

$$\mathbf{u}_2 = \mathbf{a}_2 - \text{proj}_{\mathbf{u}_1}(\mathbf{a}_2).$$

For the next vector $\mathbf{u}_3$ we want this to be orthogonal to both $\mathbf{u}_1$ and $\mathbf{u}_2$. To find $\mathbf{u}_3$ we subtract the projection of $\mathbf{a}_3$ onto $\mathbf{u}_1$ and $\mathbf{u}_2$ from $\mathbf{a}_3$. Doing similar for all vectors in A we have

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{a}_1, & \mathbf{q}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|},\\
\mathbf{u}_2 &= \mathbf{a}_2 - \text{proj}_{\mathbf{u}_1}(\mathbf{a}_2), & \mathbf{q}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|},\\
\mathbf{u}_3 &= \mathbf{a}_3 - \text{proj}_{\mathbf{u}_1}(\mathbf{a}_3) - \text{proj}_{\mathbf{u}_2}(\mathbf{a}_3), & \mathbf{q}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|},\\
&\vdots & &\vdots\\
\mathbf{u}_n &= \mathbf{a}_n - \sum_{i=1}^{n-1} \text{proj}_{\mathbf{u}_i}(\mathbf{a}_n), & \mathbf{q}_n &= \frac{\mathbf{u}_n}{\|\mathbf{u}_n\|}.
\end{aligned}$$

where $\mathbf{q}_i$ are orthonormal basis vectors. The vectors in A can be expressed using the orthonormal basis $Q = (\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n)$

$$\begin{aligned}
\mathbf{a}_1 &= (\mathbf{q}_1 \cdot \mathbf{a}_1)\mathbf{q}_1,\\
\mathbf{a}_2 &= (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1 + (\mathbf{q}_2 \cdot \mathbf{a}_2)\mathbf{q}_2,\\
\mathbf{a}_3 &= (\mathbf{q}_1 \cdot \mathbf{a}_3)\mathbf{q}_1 + (\mathbf{q}_2 \cdot \mathbf{a}_3)\mathbf{q}_2 + (\mathbf{q}_3 \cdot \mathbf{a}_3)\mathbf{q}_3,\\
&\vdots\\
\mathbf{a}_n &= \sum_{i=1}^{n} (\mathbf{q}_i \cdot \mathbf{a}_n)\mathbf{q}_i.
\end{aligned}$$

If $A = QR$ then

$$A = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 & \dots & \mathbf{q}_5 \end{pmatrix} \begin{pmatrix} \mathbf{q}_1 \cdot \mathbf{a}_1 & \mathbf{q}_1 \cdot \mathbf{a}_2 & \mathbf{q}_1 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_1 \cdot \mathbf{a}_n \\ 0 & \mathbf{q}_2 \cdot \mathbf{a}_2 & \mathbf{q}_2 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_2 \cdot \mathbf{a}_n \\ 0 & 0 & \mathbf{q}_3 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_3 \cdot \mathbf{a}_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbf{q}_n \cdot \mathbf{a}_n \end{pmatrix}.$$

To calculate the QR decomposition of an $m \times n$ matrix A using the Gram-Schmidt process we use the following steps.

For column 1 only:

$$r_{11} = \|\mathbf{a}_1\|,$$
$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{r_{11}.}$$

For all other columns $j = 2, \dots n$ calculate

$$r_{ij} = \mathbf{q}_i \cdot \mathbf{a}_j, \qquad i = 1, \dots, j-1,$$
$$\mathbf{u}_j = \mathbf{a}_j - \sum_{i=1}^{j-1} r_{ij}\mathbf{q}_i,$$
$$r_{jj} = \|\mathbf{u}_j\|,$$
$$\mathbf{q}_j = \frac{\mathbf{u}_j}{r_{jj}}.$$

Example 14

Calculate the QR decomposition of the following matrix using the Gram-Schmidt process

$$A = \begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix}.$$

Column $j = 1$:

$$r_{11} = \|\mathbf{a}_1\| = \sqrt{1^2 + 1^2 + 1^2 + 1^2} = \sqrt{4} = 2,$$
$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{r_1 1} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix}.$$

Column $j = 2$:

$$r_{12} = \mathbf{q}_1 \cdot \mathbf{a}_2 = \begin{pmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 4 \\ 4 \\ -1 \end{pmatrix} = 3,$$

$$\mathbf{u}_2 = \mathbf{a}_2 - r_{12}\mathbf{q}_1 = \begin{pmatrix} -1 \\ 4 \\ 4 \\ -1 \end{pmatrix} - 3\begin{pmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix} = \begin{pmatrix} -5/2 \\ 5/2 \\ 5/2 \\ 5/2 \end{pmatrix},$$

$$r_{22} = \|\mathbf{u}_2\| = \sqrt{(-5/2)^2 + (5/2)^2 + (5/2)^2 + (-5/2)^2} = \sqrt{25} = 5,$$

$$\mathbf{q}_2 = \frac{\mathbf{u}_2}{r_{22}} = \frac{1}{5}\begin{pmatrix} -5/2 \\ 5/2 \\ 5/2 \\ -5/2 \end{pmatrix} = \begin{pmatrix} -1/2 \\ 1/2 \\ 1/2 \\ -1/2 \end{pmatrix}.$$

Column $j = 3$:

$$r_{13} = \mathbf{q}_1 \cdot \mathbf{a}_3 = \begin{pmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ -2 \\ 2 \\ 0 \end{pmatrix} = 2,$$

$$r_{23} = \mathbf{q}_2 \cdot \mathbf{a}_3 = \begin{pmatrix} -1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ -2 \\ 2 \\ 0 \end{pmatrix} = -2,$$

$$\mathbf{u}_3 = \mathbf{u}_3 - r_{13}\mathbf{q}_1 - r_{23}\mathbf{q}_2 = \begin{pmatrix} 4 \\ -2 \\ 2 \\ 0 \end{pmatrix} - 2\begin{pmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix} + \begin{pmatrix} -1/2 \\ 1/2 \\ 1/2 \\ -1/2 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \\ 2 \\ -2 \end{pmatrix},$$

$$r_{33} = \|\mathbf{u}_3\| = \sqrt{2^2 + (-2)^2 + 2^2 + (-2)^2} = \sqrt{16} = 4,$$

$$\mathbf{q}_3 = \frac{\mathbf{u}_3}{r_{33}} = \frac{1}{4}\begin{pmatrix} 2 \\ -2 \\ 2 \\ -2 \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{pmatrix}.$$

Therefore the QR factorisation of A is

$$Q = \begin{pmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \end{pmatrix}, \qquad R = \begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix}.$$

Checking that $QR = A$

$$QR = \begin{pmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \end{pmatrix} \begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix} = A.$$

Checking that Q is orthonormal

$$Q^T Q = \begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ -1/2 & 1/2 & 1/2 & -1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \end{pmatrix} \begin{pmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = I.$$

Example 15

The function called QRdecomp below calculates the QR decomposition of the matrix `

```
function [Q, R] = QRdecomp(A)

% Calculates the QR decomposition of the m x n matrix A using the
% Gram-Schmidt process

ncols = size(A, 1);
R = zeros(ncols);
Q = zeros(size(A));

% Loop through columns of A
for j = 1 : ncols
    for i = 1 : j
        R(i, j) = dot(Q(:, i), A(:, j));
        Q(:, j) = Q(:, j) + R(i, j) * Q(:, i);
    end
    Q(:, j) = A(:, j) - Q(:, j);
    R(j, j) = norm(Q(:, j));
    Q(:, j) = Q(:, j) / R(j, j);
end

end
```

The code below uses the function QR to calculate the QR decomposition of the matrix A in example 14.

```
% Define matrix A
A = [ 1, -1, 4 ;
      1, 4, -2 ;
      1, 4, 2 ;
      1, -1, 0 ];

% Calculate QR decomposition
[Q, R] = QRdecomp(A)
```

```
Q = 4×3
    0.5000   -0.5000    0.5000
    0.5000    0.5000   -0.5000
    0.5000    0.5000    0.5000
    0.5000   -0.5000   -0.5000
R = 3×3
```

```
     2     3     2
     0     5    -2
     0     0     4
```

```
% Check that Q.R - A = 0
Q * R - A
```

```
ans = 4×3
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

```
% Check that Q^T Q = I
Q' * Q
```

```
ans = 3×3
     1     0     0
     0     1     0
     0     0     1
```

Solving systems of linear equations using QR decomposition

Given a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ then the solution can be calculated using the QR decomposition of A. Since $A = QR$ then

$$QR\mathbf{x} = \mathbf{b}$$
$$R\mathbf{x} = Q^{-1}\mathbf{b},$$

since Q is orthogonal then $Q^{-1} = Q^T$ then

$$R\mathbf{x} = Q^T\mathbf{b},$$

and since R is upper triangular the solution of $\mathbf{x}$ can be obtained through back substitution.

Example 16

Solve the following system of linear equations using QR decomposition

$$\begin{pmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 20 \\ -6 \end{pmatrix}.$$

We have seen in example 14 that the QR decomposition of the coefficient matrix is

$$Q = \begin{pmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \end{pmatrix}, \qquad R = \begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix}.$$

Solving $R\mathbf{x} = Q^T\mathbf{b}$

$$\begin{pmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ -1/2 & 1/2 & 1/2 & -1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \end{pmatrix}\begin{pmatrix} 6 \\ 8 \\ 20 \\ -6 \end{pmatrix} = \begin{pmatrix} 14 \\ 14 \\ 12 \end{pmatrix},$$

gives

$$\begin{aligned} x_3 &= \frac{12}{4} = 3, \\ x_2 &= \frac{1}{5}(14 + 2x_3) = \frac{1}{5}(14 + 2(3)) = 4, \\ x_1 &= \frac{1}{2}(14 - 3x_2 - 2x_3) = \frac{1}{2}(14 - 3(4) - 2(3)) = -2. \end{aligned}$$

So the solution is $\mathbf{x} = (-4, 4, 3)$.

Example 17

The code below solves the system of linear equations from example 16 using QR decomposition.

```
clear

% Define linear system
A = [ 1, -1, 4 ;
      1, 4, -2 ;
      1, 4, 2 ;
      1, -1, 0 ];
b = [ 6 ; 8 ; 20 ; -6 ];

% Calculate QR decomposition of A
[Q, R] = QRdecomp(A);

% Solve Rx = Q^T b using back substitution
x = back_sub(R, Q' * b);

% Output solution
for i = 1 : length(x)
    fprintf('x_%1i = %1.4f\n', i, x(i))
end
```

```
x_1 = -2.0000
x_2 = 4.0000
x_3 = 3.0000
```

Summary

- LU decomposition factorises a square matrix in the product of a lower triangular matrix L and an upper triangular matrix U.
- Crout's method is used to solve a system of linear equations using LU decomposition by apply forward and back substitution.
- Partial pivoting ensures that the pivot element has a larger absolute value than the elements in the column below the pivot. This eliminates the problems caused when the pivot element is small resulting in computational rounding errors.

- Cholesky decomposition can be applied to factorise a positive definite matrix into the product of a lower triangular matrix L and its transpose. Cholesky decomposition of a $n \times n$ matrix requires fewer operations than the equivalent LU decomposition.
- QR decomposition can be applied to factorise an $m \times n$ matrix into the product of an $m \times n$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R. QR decomposition can be calculated using the Gram-Schmidt where an orthonormal basis is determined by recursively subtracting the vector projection of non-orthogonal vectors onto known basis vectors.
- QR decomposition can be used to calculate a solution to an overdetermined system when the number of equations is bigger than the number of unknowns.
- The advantage of using decomposition methods for solving systems of linear equations is that a change in the constant values do not require a recalculation of the decomposition as opposed to Gaussian elimination.

Exercises

1. Solve the following systems of linear equations using LU decomposition by hand (without using a computer)

(a)

$$\begin{aligned} 2x_1 + 3x_2 - x_3 &= 4, \\ 4x_1 + 9x_2 - x_3 &= 18, \\ 3x_2 + 2x_3 &= 11. \end{aligned}$$

(b)

$$\begin{aligned} 3x_1 + 9x_2 + 5x_3 &= 20, \\ x_1 + 2x_2 + 2x_3 &= 3, \\ 2x_1 + 4x_2 + 5x_3 &= 4. \end{aligned}$$

(c)

$$\begin{aligned} x_1 + 3x_3 + 2x_4 &= 21, \\ 3x_1 - 2x_2 + 5x_3 + x_4 &= 28, \\ 4x_1 - x_2 - 2x_3 - 3x_4 &= -12, \\ 2x_2 + 3x_4 &= 13. \end{aligned}$$

(d)

$$\begin{aligned} x_1 + 5x_2 + 2x_3 + 2x_4 &= -10, \\ -2x_1 - 4x_2 + 2x_3 &= 10, \\ 3x_1 + x_2 - 2x_3 - x_4 &= -2, \\ -3x_1 - 3x_2 + 4x_3 - x_4 &= 4. \end{aligned}$$

2. Solve the systems from question 1 using LU decomposition with partial pivoting by hand.

3. Solve the following systems of linear equations using Cholesky decomposition by hand.

(a)

$$\begin{aligned} 16x_1 + 16x_2 + 4x_3 &= -8, \\ 16x_1 + 25x_2 + 10x_3 &= -47, \\ 4x_1 + 10x_2 + 6x_3 &= -30. \end{aligned}$$

(b)

$$\begin{aligned} 4x_1 + 2x_2 + 8x_3 &= 36, \\ 2x_1 + 17x_2 + 20x_3 &= 50, \\ 8x_1 + 20x_2 + 41x_3 &= 122. \end{aligned}$$

(c)

$$
\begin{aligned}
9x_1 - 9x_2 - 6x_4 &= 12,\\
-9x_1 + 25x_2 + 8x_3 - 10x_4 &= -116,\\
8x_2 + 8x_3 - 2x_4 &= -58,\\
-6x_1 - 10x_2 - 2x_3 + 33x_4 &= 91.
\end{aligned}
$$

(d)

$$
\begin{aligned}
x_1 + 5x_2 - x_3 + 2x_4 &= 14,\\
5x_1 + 29x_2 + 3x_3 + 12x_4 &= 82,\\
-x_1 + 3x_2 + 42x_3 - 13x_4 &= 40,\\
2x_1 + 12x_2 - 13x_3 + 39x_4 &= -34.
\end{aligned}
$$

4. Solve the following systems of linear equations using QR decomposition by hand.

(a)

$$
\begin{aligned}
x_1 + x_2 &= 9,\\
-x_1 &= -5.
\end{aligned}
$$

(b)

$$
\begin{aligned}
6x_1 + 6x_2 + x_3 &= 3,\\
3x_1 + 6x_2 + x_3 &= 0,\\
2x_1 + x_2 + x_3 &= 4.
\end{aligned}
$$

(c)

$$
\begin{aligned}
x_1 + 2x_2 + x_3 &= 1,\\
x_1 + 4x_2 + 3x_3 &= 7,\\
x_1 - 4x_2 + 6x_3 &= -6,\\
x_1 + 2x_2 + x_3 &= -1.
\end{aligned}
$$

5. Check your solutions to questions 1 to 4 using MATLAB.

Indirect Methods for Solving Systems of Linear Equations

Learning outcomes

On successful completion of this chapter readers will be able to:

- Understand the concept of an indirect method when used to solve a system of linear equations.
- Apply the Jacobi, Gauss-Seidel and SOR methods to solve a linear system.
- Use the residual to determine the accuracy of the current estimate of the solution to the linear system.
- Determine whether an indirect method is convergent for a particular linear system and analyse the theoretical rate of convergence for indirect methods.

Indirect methods

Indirect methods for solving systems of linear equations use an iterative approach to repeatedly update estimates of the exact solution to the linear system. They are called *indirect* methods since multiple applications of the method is required to calculate a solution unlike *direct* methods such as Gaussian elimination and LU decomposition which require a single application to calculate the solution. However, direct methods are inefficient for large systems of equations for which we tend to use indirect methods instead.

An indirect method for solving a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x}^{k+1} = T\mathbf{x}^k + \mathbf{c},$$

where $\mathbf{x}^k$ is the current estimate of $\mathbf{x}$, $\mathbf{x}^{k+1}$ is the improved estimate of $\mathbf{x}$, T is an **iteration matrix** and $\mathbf{c}$ is some vector. This equation is iterated updating the values of the estimates such that $\mathbf{x}^{(k)} \to \mathbf{x}$ as $k \to \infty$. Note that unlike direct methods which will calculate the exact solution, indirect only calculate an estimate (albeit very close) of the exact solution.

The Jacobi method

The Jacobi method is the simplest indirect method. Splitting the coefficient matrix A into the of elements from the lower triangular, diagonal and upper triangular parts of A to form matrices L, D and U such that $A = L + D + U$, e.g.,

$$L = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{pmatrix}, \quad D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}, \quad U = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ 0 & \cdots & 0 & 0 \end{pmatrix}.$$

Rewriting the linear system $A\mathbf{x} = \mathbf{b}$ using D, L and U gives

$$
\begin{aligned}
(L+D+U)\mathbf{x} &= \mathbf{b} \\
(L+U)\mathbf{x} + D\mathbf{x} &= \mathbf{b} \\
D\mathbf{x} &= \mathbf{b} - (L+U)\mathbf{x} \\
\mathbf{x} &= D^{-1}(\mathbf{b} - (L+U)\mathbf{x}).
\end{aligned}
$$

Let the $\mathbf{x}$ on the left-hand side be $\mathbf{x}^{(k+1)}$ and the $\mathbf{x}$ on the right-hand side be $\mathbf{x}^{(k)}$ then

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L+U)\mathbf{x}), \qquad (1)$$

and writing this out for each element we have

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{N} a_{ij}x_j^{(k)}\right), \qquad i = 1, 2, \ldots, n. \qquad (2)$$

The residual

The Jacobi method is applied by iterating equation (2) until the solution $\mathbf{x}^{(k+1)}$ is accurate enough for our needs. Since we do not know what the exact solution is, we can quantity the accuracy of an estimate by using the **residual** which is defined as

$$\mathbf{r} = \mathbf{b} - A\mathbf{x}^{(k)}, \qquad (3)$$

so that as $\mathbf{x}^{(k)} \to \mathbf{x}$, $\mathbf{r} \to \mathbf{0}$. The convergence criteria used is

$$|\mathbf{r}| < tol,$$

where tol is some small number. The smaller tol is, the closer $\mathbf{x}^{(k)}$ is to the exact solution but this will require more iterations. In practice a compromise is made between the accuracy required and the computational resources available. Typical values of tol are around 10^{-4} or maybe even 10^{-6}.

Example 1

Calculate the first iteration of the Jacobi method to solve the following system of linear equations and calculate the norm of the residual.

$$
\begin{aligned}
4x_1 + 3x_2 &= -2, \\
3x_1 + 4x_2 - x_3 &= -8, \\
-x_2 + 4x_3 &= 14.
\end{aligned}
$$

The Jacobi iterations are

$$
\begin{aligned}
x_1^{(k+1)} &= \frac{1}{4}(-2 - 3x_2^{(k)}), \\
x_2^{(k+1)} &= \frac{1}{4}(-8 - 3x_1^{(k)} + x_3^{(k)}), \\
x_3^{(k+1)} &= \frac{1}{4}(14 + x_2^{(k)}).
\end{aligned}
$$

Using starting values of $\mathbf{x}^{(0)} = (0, 0, 0)^T$ the first iteration is

$$x_1^{(1)} = \frac{1}{4}(-2 - 3(0)) = -0.5,$$
$$x_2^{(1)} = \frac{1}{4}(-8 - 3(0) + 0) = -2,$$
$$x_3^{(1)} = \frac{1}{4}(14 + 0) = 3.5.$$

Calculate the residual using equation (2)

$$\mathbf{r}^{(1)} = \mathbf{b} - A\mathbf{x}^{(1)} = \begin{pmatrix} -2 \\ -8 \\ 14 \end{pmatrix} - \begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} -0.5 \\ -2 \\ 3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ -2 \end{pmatrix},$$

and the norm of the residual is $|\mathbf{r}^{(1)}| = \sqrt{6^2 + 5^2 + (-2)^2} = 8.06226..$

Example 2

Write a MATLAB program to solve the system of linear equations from example 1 using the Jacobi method ceasing iterations when $|\mathbf{r}| < 10^{-4}$.

The function called `jacobi` below solves a linear system of equations defined by the arrays `A` and `b`. Iterations cease when $|\mathbf{r}| < tol$ or `maxiter` is exceeded.

```
function x = jacobi(A, b, maxiter, tol)

% Calculates the solution to the system of linear equations Ax = b using
% the Jacobi method

% Initialise solution array
N = length(b);
x = zeros(maxiter + 1, N);

% Iteration loop
for k = 1 : maxiter

    % Calculate Jacobi method
    for i = 1 : N

        % Calculate sum
        for j = 1 : N
            if i ~= j
                x(k+1, i) = x(k+1, i) + A(i, j) * x(k, j);
            end
        end

        % Calculate new estimate of x(i)
        x(k+1, i) = (b(i) - x(k+1, i)) / A(i, i);
    end

    % Calculate norm of the residual
    r = norm(b - A * x(k+1, :)');

    % Check for convergence
    if r < tol
        break
    end
```

```
end

% Trim x array
x(k+2:end, :) = [];

end
```

The program below uses the function `jacobi` to solve the system of linear equations from example 1.

```
% Define linear system
A = [ 4, 3, 0 ; 3, 4, -1 ; 0, -1, 4 ];
b = [ -2 ; -8 ; 14 ];

% Solve linear system using the Jacobi method
x = jacobi(A, b, 100, 1e-4);

% Ouput solution table
N = length(b);
table = '  k      ';
for i = 1 : N
    table = [ table, sprintf('x_%1i       ', i)];
end
table = [ table, sprintf('|r|\n%s\n', repmat('-', 1, 4 + (N + 1) * 10)) ];
for k = 1 : size(x, 1)
    table = [ table, sprintf('%4i', k - 1) ];
    table = [ table, sprintf('%10.5f', x(k, :)) ];
    table = [ table, sprintf('%10.5f\n', norm(b - A * x(k,:)')) ];
end
fprintf(table)
```

```
  k       x_1       x_2       x_3       |r|
-----------------------------------------------
  0   0.00000   0.00000   0.00000  16.24808
  1  -0.50000  -2.00000   3.50000   8.06226
  2   1.00000  -0.75000   3.00000   6.37377
  3   0.06250  -2.00000   3.31250   5.03891
  4   1.00000  -1.21875   3.00000   3.98361
  5   0.41406  -2.00000   3.19531   3.14932
  6   1.00000  -1.51172   3.00000   2.48976
  7   0.63379  -2.00000   3.12207   1.96832
  8   1.00000  -1.69482   3.00000   1.55610
  9   0.77112  -2.00000   3.07629   1.23020
 10   1.00000  -1.80927   3.00000   0.97256
 11   0.85695  -2.00000   3.04768   0.76888
 12   1.00000  -1.88079   3.00000   0.60785
 13   0.91059  -2.00000   3.02980   0.48055
 14   1.00000  -1.92549   3.00000   0.37991
 15   0.94412  -2.00000   3.01863   0.30034
 16   1.00000  -1.95343   3.00000   0.23744
 17   0.96508  -2.00000   3.01164   0.18771
 18   1.00000  -1.97090   3.00000   0.14840
 19   0.97817  -2.00000   3.00728   0.11732
 20   1.00000  -1.98181   3.00000   0.09275
 21   0.98636  -2.00000   3.00455   0.07333
 22   1.00000  -1.98863   3.00000   0.05797
 23   0.99147  -2.00000   3.00284   0.04583
 24   1.00000  -1.99289   3.00000   0.03623
 25   0.99467  -2.00000   3.00178   0.02864
 26   1.00000  -1.99556   3.00000   0.02264
 27   0.99667  -2.00000   3.00111   0.01790
 28   1.00000  -1.99722   3.00000   0.01415
 29   0.99792  -2.00000   3.00069   0.01119
 30   1.00000  -1.99827   3.00000   0.00885
```

31	0.99870	−2.00000	3.00043	0.00699
32	1.00000	−1.99892	3.00000	0.00553
33	0.99919	−2.00000	3.00027	0.00437
34	1.00000	−1.99932	3.00000	0.00346
35	0.99949	−2.00000	3.00017	0.00273
36	1.00000	−1.99958	3.00000	0.00216
37	0.99968	−2.00000	3.00011	0.00171
38	1.00000	−1.99974	3.00000	0.00135
39	0.99980	−2.00000	3.00007	0.00107
40	1.00000	−1.99983	3.00000	0.00084
41	0.99988	−2.00000	3.00004	0.00067
42	1.00000	−1.99990	3.00000	0.00053
43	0.99992	−2.00000	3.00003	0.00042
44	1.00000	−1.99994	3.00000	0.00033
45	0.99995	−2.00000	3.00002	0.00026
46	1.00000	−1.99996	3.00000	0.00021
47	0.99997	−2.00000	3.00001	0.00016
48	1.00000	−1.99997	3.00000	0.00013
49	0.99998	−2.00000	3.00001	0.00010
50	1.00000	−1.99998	3.00000	0.00008

So the Jacobi method took 50 iterations to converge to the solution $x_1 = 1.0000,\ x_2 = -2.0000$ and $x_3 = 3.0000$ correct to 4 decimal places.

The Gauss-Seidel method

The speed of convergence for the Jacobi method can be improved by using the values of $x_j^{(k+1)}$ where $j < i$ to calculate $x_i^{(k+1)}$. This leads to the Gauss-Seidel method. Writing the coefficient matrix of a system of linear equations $A\mathbf{x} = \mathbf{b}$ as $A = L + D + U$

$$\begin{aligned}(L + D + U)\mathbf{x} &= \mathbf{b}\\ D\mathbf{x} &= \mathbf{b} - L\mathbf{x} - U\mathbf{x}\\ \mathbf{x} &= D^{-1}(\mathbf{b} - L\mathbf{x} - U\mathbf{x}).\end{aligned}$$

Since L is a lower triangular matrix the values of $x_i^{(k+1)}$ can be calculated sequentially using forward substitution so the iterative form of the Gauss-Seidel method is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \qquad i = 1, 2, \ldots, n. \tag{4}$$

Example 3

Calculate the first iteration of the Gauss-Seidel method and the norm of the residual for the system of linear equations from example 1

$$\begin{aligned}4x_1 + 3x_2 &= -2,\\ 3x_1 + 4x_2 - x_3 &= -8,\\ -x_2 + 4x_3 &= 14.\end{aligned}$$

The Gauss-Seidel iterations are

$$x_1^{(k+1)} = \frac{1}{4}(-2 - 3x_2^{(k)}),$$
$$x_2^{(k+1)} = \frac{1}{4}(-8 - 3x_1^{(k+1)} + x_3^{(k)}),$$
$$x_3^{(k+1)} = \frac{1}{4}(14 + x_2^{(k+1)}).$$

Using starting values of $\mathbf{x}^{(0)} = (0, 0, 0)^T$ the first iteration is

$$x_1^{(1)} = \frac{1}{4}(-2 - 3(0)) = -0.5,$$
$$x_2^{(1)} = \frac{1}{4}(-8 - 3(-0.5) + 0) = -1.625$$
$$x_3^{(1)} = \frac{1}{4}(14 - 1.625) = 3.09375.$$

Calculate the residual

$$\mathbf{r}^{(1)} = \mathbf{b} - A\mathbf{x}^{(1)} = \begin{pmatrix} -2 \\ -8 \\ 14 \end{pmatrix} - \begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} -0.5 \\ -1.625 \\ 3.09375 \end{pmatrix} = \begin{pmatrix} 4.875 \\ 3.09375 \\ 0 \end{pmatrix},$$

and the norm of the residual is $|\mathbf{r}^{(1)}| = \sqrt{4.875^2 + 3.09375^2 + 0^2} = 5.77381$.

Example 4

Write a MATLAB program to solve the system of linear equations from example 1 using the Gauss-Seidel method.

The function called `gauss_seidel` below solves a linear system of equations defined by the arrays `A` and `b`. Iterations cease when $|\mathbf{r}| < tol$ or `maxiter` is exceeded.

```
function x = gauss_seidel(A, b, maxiter, tol)

% Calculates the solution to the system of linear equations Ax = b using
% the Gauss-Seidel method

% Initialise solution array
N = length(b);
x = zeros(maxiter + 1, N);

% Iteration loop
for k = 1 : maxiter

    % Calculate Gauss-Seidel method
    for i = 1 : N

        % Calculate sum
        for j = 1 : N
            if j < i
                x(k+1, i) = x(k+1, i) + A(i, j) * x(k+1, j);
            elseif j > i
                x(k+1, i) = x(k+1, i) + A(i, j) * x(k, j);
```

```
            end
        end

        % Calculate new estimate of x(i)
        x(k+1, i) = (b(i) - x(k+1, i)) / A(i, i);
    end

    % Calculate norm of the residual
    r = norm(b - A * x(k+1, :)');

    % Check for convergence
    if r < tol
        break
    end

end

% Trim x array
x(k+2:end, :) = [];

end
```

The program below uses the function `gauss_seidel` to solve the system of linear equations from example 1.

```
% Define linear system
A = [ 4, 3, 0 ; 3, 4, -1 ; 0, -1, 4 ];
b = [ -2 ; -8 ; 14 ];

% Solve linear system using the Jacobi method
x = gauss_seidel(A, b, 100, 1e-4);

% Ouput solution table
N = length(b);
table = '  k      ';
for i = 1 : N
    table = [ table, sprintf('x_%1i       ', i)];
end
table = [ table, sprintf('|r|\n%s\n', repmat('-', 1, 4 + (N + 1) * 10)) ];
for k = 1 : size(x, 1)
    table = [ table, sprintf('%4i', k - 1) ];
    table = [ table, sprintf('%10.5f', x(k, :)) ];
    table = [ table, sprintf('%10.5f\n', norm(b - A * x(k,:)')) ];
end
fprintf(table)
```

```
  k      x_1       x_2       x_3       |r|
-------------------------------------------
  0   0.00000   0.00000   0.00000  16.24808
  1  -0.50000  -1.62500   3.09375   5.77381
  2   0.71875  -1.76562   3.05859   0.42334
  3   0.82422  -1.85352   3.03662   0.26459
  4   0.89014  -1.90845   3.02289   0.16537
  5   0.93134  -1.94278   3.01431   0.10335
  6   0.95708  -1.96424   3.00894   0.06460
  7   0.97318  -1.97765   3.00559   0.04037
  8   0.98324  -1.98603   3.00349   0.02523
  9   0.98952  -1.99127   3.00218   0.01577
 10   0.99345  -1.99454   3.00136   0.00986
 11   0.99591  -1.99659   3.00085   0.00616
 12   0.99744  -1.99787   3.00053   0.00385
 13   0.99840  -1.99867   3.00033   0.00241
```

```
14   0.99900  -1.99917   3.00021   0.00150
15   0.99938  -1.99948   3.00013   0.00094
16   0.99961  -1.99967   3.00008   0.00059
17   0.99976  -1.99980   3.00005   0.00037
18   0.99985  -1.99987   3.00003   0.00023
19   0.99990  -1.99992   3.00002   0.00014
20   0.99994  -1.99995   3.00001   0.00009
```

Note that the Gauss-Seidel method took 20 iterations to achieve convergence to $tol = 10^{-4}$ whereas the Jacobi method took 50 iterations to achieve the same accuracy.

Convergence of direct methods

We have seen that both the Jacobi and Gauss-Seidel method converge to the solution to the example system of linear equations and that the Gauss-Seidel method converges at a faster rate. Direct methods will not be convergent for all linear systems and we tell if a method will be convergent using the theorem below

We can examine the convergence of an indirect method by considering the iteration of the error vector $\mathbf{e}^{(k)}$

$$\mathbf{e}^{(k+1)} = T\mathbf{e}^{(k)},$$

if $\mathbf{v}_i$ are the eigenvectors of the iteration matrix T with corresponding eigenvalues λ_i then for the first error we can write

$$\mathbf{e}^{(0)} = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \cdots + \alpha_n\mathbf{v}_n = \sum_{i=1}^{n} \alpha_i\mathbf{v}_i.$$

where α_i are scalars. Applying an iteration matrix to $\mathbf{e}^{(k)}$

$$\begin{aligned}
\mathbf{e}^{(1)} &= T\mathbf{e}^{(0)} = T\left(\sum_{i=1}^{n} \alpha_i\mathbf{v}_i\right) = \sum_{i=1}^{n} \alpha_i T\mathbf{v}_i = \sum_{i=1}^{n} \alpha_i\lambda_i\mathbf{v}_i, \\
\mathbf{e}^{(2)} &= T\mathbf{e}^{(1)} = T\left(\sum_{i=1}^{n} \alpha_i\lambda_i\mathbf{v}_i\right) = \sum_{i=1}^{n} \alpha_i\lambda_i T\mathbf{v}_i = \sum_{i=1}^{n} \alpha_i\lambda_i^2\mathbf{v}_i, \\
&\vdots \\
\mathbf{e}^{(k+1)} &= \sum_{i=1}^{n} \alpha_i\lambda_i^k\mathbf{v}_i.
\end{aligned}$$

If $|\lambda_1| > \lambda_i$ for $i = 2, 3 \ldots, n$ then

$$\mathbf{e}^{(k+1)} = \alpha_1\lambda_1^{k+1}\mathbf{v}_1 + \sum_{i=2}^{n} \alpha_i\lambda_i^{k+1}\mathbf{v}_i = \lambda_1^{k+1}\left(\alpha_1\mathbf{v}_1 + \sum_{i=2}^{n} \alpha_i\mathbf{v}_i\left(\frac{\lambda_i}{\lambda_1}\right)^{(k+1)}\right),$$

and since $\left(\frac{\lambda_i}{\lambda_1}\right)^{(k+1)}$ so

$$\lim_{k\to\infty} \mathbf{e}^{(k+1)} = \alpha_1\lambda_1^{(k+1)}\mathbf{v}_1.$$

This means that the error varies by a factor of $\lambda_1^{(k+1)}$ where λ_1 is the largest eigenvalue of T which is also known as the spectral radius and denoted by $\rho(T)$. The spectral radius gives us the following information about an indirect method

- If $\rho(T) > 1$ then the errors will increase over each iteration, therefore for an indirect method to converge to the solution we require $\rho(T) < 1$.
- The smaller the value of $\rho(T)$ the faster the errors will tend to zero.

Example 5

Show that the Jacobi and Gauss-Seidel methods are convergent of the system of linear equations from example 1.

Recall that the matrix form of the Jacobi method is equation (1)

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L+U)\mathbf{x}^{(k)}) = -D^{-1}(L+U)\mathbf{x}^{(k)} + D^{-1}\mathbf{b},$$

so the iteration matrix for the Jacobi method is

$$T_J = -D^{-1}(L+U). \qquad (5)$$

The iteration matrix for the Gauss-Seidel method can be found by rearranging $(L+D+U)\mathbf{x} = \mathbf{b}$

$$\begin{aligned}(L+D+U)\mathbf{x} &= \mathbf{b}\\ (L+D)\mathbf{x} &= \mathbf{b} - U\mathbf{x}\\ \mathbf{x} &= -(L+D)^{-1}U\mathbf{x} + (L+D)^{-1}\mathbf{x}\end{aligned}$$

So the Gauss-Seidel method is

$$\mathbf{x}^{(k+1)} = -(L+D)^{-1}U\mathbf{x}^{(k)} + (L+D)^{-1}\mathbf{x}^{(k+1)},$$

and the iteration matrix for the Gauss-Seidel method is

$$T_{GS} = -(L+D)^{-1}U. \qquad (6)$$

The code below calculates the spectral radius of T_J and T_{GS}.

```
clear

% Define coefficient matrix
A = [ 4, 3, 0 ; 3, 4, -1 ; 0, -1, 4 ];

% Calculate L, D and U
L = tril(A, -1);
U = triu(A, 1);
D = A - L - U;

% Calculate iteration matrices
TJ = -inv(D) * (L + U);
TGS = -inv(L + D) * U;

% Calculate spectral radii of TJ and TGS
rho_TJ = max(abs(eig(TJ)));
rho_TGS = max(abs(eig(TGS)));
```

```
% Output results
fprintf('rho(T_J) = %1.4f\nrho(T_GS) = %1.4f', rho_TJ, rho_TGS)
```

```
rho(T_J) = 0.7906
rho(T_GS) = 0.6250
```

So $\rho(T_J) = 0.7906$ and $\rho(T_{GS}) = 0.6250$ which are both less than 1 so both of these methods is convergent for this system. Furthermore, the Gauss-Seidel method will converge faster than the Jacobi method since it has a smaller spectral radius.

The Successive Over Relaxation (SOR) method

The Successive Over Relaxation (SOR) method improves on the convergence rate of the Gauss-Seidel method by applying a weighting factor to the updated estimates to increase (or decrease in the case of oscillatory convergence) the effect of the change.

Let ω be a **relaxation parameter** in the range $[0, 2]$ then the SOR method is

$$\mathbf{x}^{(k+1)} = (1-\omega)\mathbf{x}^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\mathbf{x}_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\mathbf{x}_j^{(k)}\right). \tag{7}$$

When $\omega < 1$ the coefficient of the first term in equation (7) is multiplied by a number greater than 1 and the second term (which is the Gauss-Seidel method) is multiplied by a number less than 1 therefore reducing the change in the value of the estimate from one iteration to the next. This will speed up the rate of convergence when the estimates oscillate about the exact solution.

When $\omega > 1$ the coefficient of the first term is multiplied by a number less than 1 and the second term by a number greater than 1 therefore increasing the change in the estimates. This will speed up the rate of convergence when the estimates converge monotonically.

Optimum value of the relaxation parameter

The optimum value of ω will be the one that minimises the spectral radius of the iteration matrix. The iteration matrix for the SOR method is derived by writing the coefficient matrix of the linear system $A\mathbf{x} = \mathbf{b}$ using

$$A = L + \left(1 - \frac{1}{\omega}\right)D + \frac{1}{\omega}D + U.$$

Substituting into the linear system and rearranging

$$\begin{aligned}
\left(L + \left(1 - \frac{1}{\omega}\right)D + \frac{1}{\omega}D + U\right)\mathbf{x} &= \mathbf{b} \\
(D + \omega L)\mathbf{x} + ((\omega - 1)D + \omega U)\mathbf{x} &= \omega\mathbf{b} \\
(D + \omega L)\mathbf{x} &= ((1-\omega)D - \omega U)\mathbf{x} + \omega\mathbf{b} \\
\mathbf{x} &= (D + \omega L)^{-1}((1-\omega)D - \omega U)\mathbf{x} + (D + \omega L)^{-1}\omega\mathbf{b}.
\end{aligned}$$

So the matrix form of the SOR method is

$$\mathbf{x}^{(k+1)} = (D + \omega L)^{-1}((1-\omega)D - \omega U)\mathbf{x}^{(k)} + (D + \omega L)^{-1}\omega\mathbf{b},$$

and the iteration matrix is

$$T_{SOR} = (D + \omega L)^{-1}((1 - \omega)D - \omega U). \qquad (8)$$

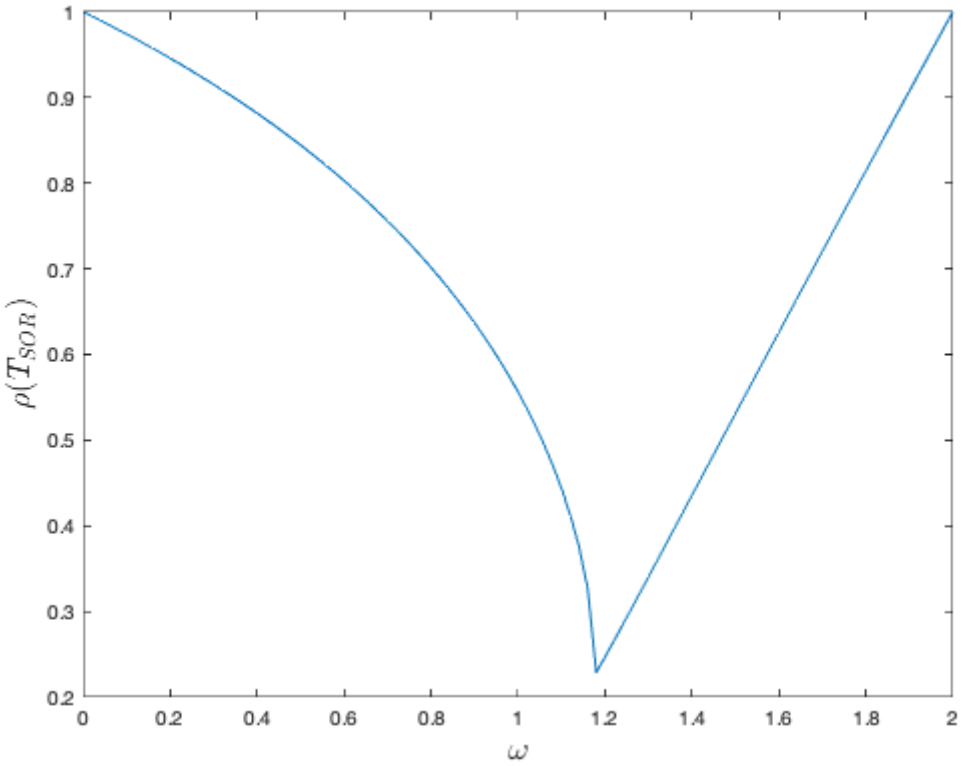
Equation (8) is a function of ω so the spectral radius of T_{SOR} will depend on ω.

Example 6

Determine the optimum relaxation parameter for the SOR when applied to the linear system from example 1.

The code below calculates the spectral radius of the SOR method for a range of ω values and plots them.

```
clear

% Define coefficient matrix
A = [ 4, 3, -1 ; 3, 4, -1 ; -1, -1, 4 ];

% Calculate L, D and U
L = tril(A, -1);
U = triu(A, 1);
D = A - L - U;

% Calculate spectral radius for a range of omega values
omega = linspace(0, 2, 101);
rho_TSOR = zeros(size(omega));
for i = 1 : length(omega)
    TSOR = inv(D + omega(i) * L) * ((1 - omega(i)) * D - omega(i) * U);
    rho_TSOR(i) = max(abs(eig(TSOR)));
end

% Plot spectral radius against omega
plot(omega, rho_TSOR)

xlabel('$\omega$', 'FontSize', 16', 'Interpreter', 'latex')
ylabel('$\rho(T_{SOR})$', 'FontSize', 16, 'Interpreter', 'latex')
```



The value of $\rho(T_{SOR})$ is a minimum when $\omega \approx 1.2$ so this is the optimum value.

Theorem (optimum relaxation parameters for a positive definite system)

If a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ has a positive definite coefficient matrix A with all real eigenvalues then the optimum relaxation parameter for the SOR method can be calculated using

$$\omega_{opt} = 1 + \left(\frac{\rho(T_J)}{1 + \sqrt{1 - \rho(T_J)^2}} \right)^2,$$

where T_J is the iteration matrix for the Jacobi method.

Example 7

Determine the optimum value of the relaxation parameter ω for the linear system from example 1.

Checking that A is positive definite

```
A = [ 4, 3, -1 ; 3, 4, -1 ; -1, -1, 4 ];
eig(A)
```

```
ans = 3×1
    1.0000
    3.4384
    7.5616
```

So $\lambda_1 = 1$, $\lambda_2 = 3.4384$ and $\lambda_3 = 7.5616$ which are all real and A is symmetric so A is a positive definite matrix.

We saw in example 6 that $\rho(T_J) = 0.7906$, so the optimum value of ω is

$$\omega_{opt} = 1 + \left(\frac{0.7906}{1 + \sqrt{1 - 0.7906^2}} \right)^2 = 1.2404.$$

Example 8

Calculate the first iteration of the SOR method using $\omega = 1.24$ and the norm of the residual for the system of linear equations from example 1

$$\begin{aligned} 4x_1 + 3x_2 &= -2, \\ 3x_1 + 4x_2 - x_3 &= -8, \\ -x_2 + 4x_3 &= 14. \end{aligned}$$

The SOR iterations are

$$\begin{aligned} x_1^{(k+1)} &= (1 - 1.24)x_1^{(k)} + \frac{1.24}{4}(-2 - 3x_2^{(k)}), \\ x_2^{(k+1)} &= (1 - 1.24)x_2^{(k)} + \frac{1.24}{4}(-8 - 3x_1^{(k+1)} + x_3^{(k)}), \\ x_3^{(k+1)} &= (1 - 1.24)x_3^{(k)} + \frac{1.24}{4}(14 + x_2^{(k+1)}). \end{aligned}$$

Using starting values of $\mathbf{x}^{(0)} = (0, 0, 0)^T$ the first iteration is

$$\begin{aligned} x_1^{(k+1)} &= (1 - 1.24)(0) + \frac{1.24}{4}(-2 - 3(0)) = -0.62, \\ x_2^{(k+1)} &= (1 - 1.24)(0) + \frac{1.24}{4}(-8 - 3(-0.62) + 0) = -1.9034, \\ x_3^{(k+1)} &= (1 - 1.24)(0) + \frac{1.24}{4}(14 - 1.9034) = 3.74995. \end{aligned}$$

Calculate the residual

$$\mathbf{r}^{(1)} = \mathbf{b} - A\mathbf{x}^{(1)} = \begin{pmatrix} -2 \\ -8 \\ 14 \end{pmatrix} - \begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} -0.62 \\ -1.9034 \\ 3.74995 \end{pmatrix} = \begin{pmatrix} 6.1902 \\ 5.2235 \\ -2.9032 \end{pmatrix}$$

and the norm of the residual is $|\mathbf{r}^{(1)}| = \sqrt{6.1902^2 + 5.2235^2 + (-2.9032)^2} = 8.60421$.

Example 9

Write a MATLAB program to solve the system of linear equations from example 1 using the SOR method with $\omega = 1.24$.

The function called `sor` below solves a linear system of equations defined by the arrays `A` and `b` using the SOR method. Iterations cease when $|\mathbf{r}| < tol$ or `maxiter` is exceeded.

```
function x = sor(A, b, omega, maxiter, tol)

% Calculates the solution to the system of linear equations Ax = b using
% the SOR method

% Initialise solution array
N = length(b);
x = zeros(maxiter + 1, N);

% Iteration loop
for k = 1 : maxiter

    % Calculate SOR method
    for i = 1 : N

        % Calculate sum
        for j = 1 : N
            if j < i
                x(k+1, i) = x(k+1, i) + A(i, j) * x(k+1, j);
            elseif j > i
                x(k+1, i) = x(k+1, i) + A(i, j) * x(k, j);
            end
        end

        % Calculate new estimate of x(i)
        x(k+1, i) = (1 - omega) * x(k, i) + omega * (b(i) - x(k+1, i)) / A(i, i);
    end

    % Calculate norm of the residual
    r = norm(b - A * x(k+1, :)');

    % Check for convergence
    if r < tol
        break
    end

end

% Trim x array
x(k+2:end, :) = [];

end
```

The program below uses the function `sor` to solve the system of linear equations from example 1.

```
% Define linear system
A = [ 4, 3, 0 ; 3, 4, -1 ; 0, -1, 4 ];
b = [ -2 ; -8 ; 14 ];

% Solve linear system using the SOR method
x = sor(A, b, 1.24, 100, 1e-4);

% Ouput solution table
N = length(b);
table = '  k      ';
```

```
for i = 1 : N
    table = [ table, sprintf('x_%1i`       ', i)];
end
table = [ table, sprintf('|r|\n%s\n', repmat('-', 1, 4 + (N + 1) * 10)) ];
for k = 1 : size(x, 1)
    table = [ table, sprintf('%4i', k - 1) ];
    table = [ table, sprintf('%10.5f', x(k, :)) ];
    table = [ table, sprintf('%10.5f\n', norm(b - A * x(k,:)')) ];
end
fprintf(table)
```

```
  k      x_1       x_2       x_3       |r|
-----------------------------------------------
  0   0.00000   0.00000   0.00000  16.24808
  1  -0.62000  -1.90340   3.74995   8.60421
  2   1.29896  -2.06874   2.79870   1.48306
  3   0.99217  -2.03863   3.03634   0.31850
  4   1.03780  -2.01462   2.98675   0.13283
  5   1.00452  -2.00481   3.00169   0.01419
  6   1.00338  -2.00147   2.99914   0.01066
  7   1.00055  -2.00043   3.00007   0.00118
  8   1.00027  -2.00012   2.99994   0.00080
  9   1.00005  -2.00003   3.00000   0.00011
 10   1.00002  -2.00001   3.00000   0.00006
```

Note that the SOR method took 10 iterations to achieve convergence to $tol = 10^{-4}$ whereas the Gauss-Seidel and Jacobi methods took 20 and 50 iterations respectively to achieve the same accuracy.

Summary

- Indirect methods use an iterative approach to improve an estimate of the solution to a system of linear equations. The methods are iterated until the estimates have achieved the required accuracy.
- The Jacobi method is uses information from the previous iteration only to update the estimates.
- The Gauss-Seidel method uses values of estimates already calculated in a given iteration to update the estimates. This means that the Gauss-Seidel method will converge to a solution faster than the Jacobi method.
- An indirect method will converge to the exact solution if the value of the spectral radius (the largest absolute eigenvalue) of the iteration matrix for a linear system is less than 1.
- The smaller the value of the spectral radius, the faster the method will converge to the exact solution.
- The SOR method uses a relaxation parameter to adjust how much estimates will change over a single iteration. The value of the relaxation parameter is chosen to minimise the spectral radius.

Exercises

1. Using a pen and calculator, calculate the first 2 iterations of the Jacobi method for solving the system of linear equations below. Use starting values of $x_i^{(0)} = 0$ and work to 4 decimal places.

$$\begin{aligned} 4x_1 + x_2 - x_3 + x_4 &= 14, \\ x_1 + 4x_2 - x_3 - x_4 &= 10, \\ -x_1 - x_2 + 5x_3 + x_4 &= -15, \\ x_1 - x_2 + x_3 + 3x_4 &= 3. \end{aligned}$$

2. Repeat question 1 using the Gauss-Seidel method.

3. Repeat question 1 using the SOR method using the optimum value for the relaxation parameter (use MATLAB to determine the spectral radius of T_J).

4. Which method would you expect to converge to the solution with the fewest iterations? What quantitative evidence do you have to support your conclusion?

5. Write a MATLAB program to calculate the solution to questions 1 to 3 using $tol = 10^{-5}$. How many iterations did each of the three methods take to converge to the solution?

6. A linear system has the following coefficient matrix. What is the largest value that α can be in order for the Jacobi method to be convergent?

$$A = \begin{pmatrix} 2 & 1 \\ \alpha & 2 \end{pmatrix}$$

7. A linear system has the following coefficient matrix. Is the Jacobi method convergent for this system? If not, can the system be altered so that the Jacobi method is convergent?

$$A = \begin{pmatrix} 1 & 3 & 1 \\ 2 & 1 & 0 \\ 1 & 1 & 4 \end{pmatrix}$$