

Runge Kutta Method for Solving ODE

T.B.Co, 2018

1. Problem setup

- There are n first order differential equations written in the form

$$\frac{dx_i}{dt} = f_i(t, x_1, \dots, x_n) \quad ; \quad i = 1, \dots, n \quad (1)$$

in n dynamic variables: x_1, \dots, x_n (also known as “**states**”) and independent variable t .

- These functions can then be arranged as column vector \mathbf{f} . Thus, the set of ODEs can be written as¹

$$\frac{d}{dt} \mathbf{x} = \mathbf{f}(t, \mathbf{x}) \quad (2)$$

(also known as the “**state space formulation**”)

- If all the conditions are given at $t = 0$: $\mathbf{x}(0) = \mathbf{x}_0$, then it is referred to as “**initial value problem**”. Otherwise, if some conditions are given at other points, say $t = t_a$, then the problem becomes a “**multi-point boundary value problem**”.²

2. Main Method

Numerical methods³ for solving initial value problems essentially invokes a “marching forward” approach. For a uniform time-increment Δt , then at some $t = t_k = k\Delta t$, the values at $\mathbf{x}(t_k) = \mathbf{x}_k$ is moved forward by some incremental change

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta_k \quad (3)$$

For the simplest case, the algorithm known as Euler method, this update is given by

$$(\Delta_k)_{Euler} = \mathbf{f}(t_k, \mathbf{x}_k)\Delta t \quad (4)$$

¹ A high order differential equation can often be put into the needed form of (2). See appendix A.

² We will focus only on solving the initial value problems (IVP).

³ We will restrict our approach only to methods known as “explicit methods” that yield the form in (3).

A significant improvement can be achieved by having some additional calculation at intermediate values between t_k and t_{k+1} , i.e. before implementing the change to \mathbf{x}_k . One of the most popular extension is known as the 4th order Runge-Kutta method.⁴ First, we evaluate four intermediate update calculations:

$$\begin{aligned}
 \delta_1 &= \Delta t \mathbf{f}(t_k, \mathbf{x}_k) \\
 \delta_2 &= \Delta t \mathbf{f}\left(\left[t_k + \frac{1}{2}\Delta t\right], \left[\mathbf{x}_k + \frac{1}{2}\delta_1\right]\right) \\
 \delta_3 &= \Delta t \mathbf{f}\left(\left[t_k + \frac{1}{2}\Delta t\right], \left[\mathbf{x}_k + \frac{1}{2}\delta_2\right]\right) \\
 \delta_4 &= \Delta t \mathbf{f}\left([t_k + \Delta t], [\mathbf{x}_k + \delta_3]\right)
 \end{aligned} \tag{5}$$

Then we combine the four updates via the following weighted average:

$$(\Delta_k)_{RK} = \frac{1}{6}(\delta_1 + 2\delta_2 + 2\delta_3 + \delta_4) \tag{6}$$

and,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\Delta_k)_{RK}$$

Thus, starting with $\mathbf{x}_k = \mathbf{x}_0$ at $t = 0$, the update will yield \mathbf{x}_1 corresponding to $t = t_1 = t_0 + \Delta t$. Then with $\mathbf{x}_k = \mathbf{x}_1$, the update will yield \mathbf{x}_2 corresponding to $t = t_2 = t_1 + \Delta t$, and so forth, until the final time $t = t_{final}$ is reached.⁵

⁴ The term “fourth order” refers to the fact that the error will be in the order of magnitude of Δt^4 .

⁵ Note that the requirement of uniform Δt can easily be removed by simply replacing Δt by Δt_k in (5), which means the forward marching increment can either be lengthened or shortened.

3. Matlab Implementation

```
function [tsoln,xsoln] = runge_kutta(fcn,tinit,tfinal,xinit,npts)
%
% function [tsoln,xsoln] = runge_kutta(fcn,tinit,tfinal,xinit,npts)
% =====
% 4th order Runge Kutta
% input: fcn=function handle for derivatives
%        tinit,tfinal= initial and final time
%        xinit= initial value
%        (optional) dt= time increment (default=200)

if nargin<5
    npts = 200;
end
dt = (tfinal-tinit)/(npts-1);
tsoln = linspace(tinit,tfinal,npts)';
Nt = length(tsoln);
n = length(xinit);
xsoln = zeros(Nt,n);
t = tinit;
x = xinit(:);
tsoln(1) = tinit;
xsoln(1,:) = x';

for k=2:Nt
    delta1 = dt*feval(fcn,t,x);
    delta2 = dt*feval(fcn,t+dt/2,x+delta1/2);
    delta3 = dt*feval(fcn,t+dt/2,x+delta2/2);
    delta4 = dt*feval(fcn,t+dt,x+delta3);
    dx = (delta1 + 2*delta2 + 2*delta3 + delta4)/6;
    x = x + dx;
    t = t + dt;
    tsoln(k) = t;
    xsoln(k,:) = x';
end
end
```

4. Test Example: Multiple Linear ODE

Consider the 5th order linear ODE given by:

$$\frac{d}{dt} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} -0.6 & -3.1 & -3.2 & -0.5 & -1.2 \\ -0.1 & 0.5 & 1.3 & 0.9 & -1.3 \\ -2.1 & -0.9 & -2.0 & -0.1 & -2.8 \\ 0.0 & -3.6 & -4.4 & -1.9 & -0.9 \\ -0.6 & 1.5 & 1.3 & 0.4 & -1.3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} + \begin{pmatrix} 0.0 \\ -0.2 \\ -0.1 \\ 0.1 \\ 0.0 \end{pmatrix} \quad (7)$$

Subject to the initial condition:

$$\mathbf{x}(0) = \mathbf{x}_0 = \begin{pmatrix} -4 \\ 3 \\ -2 \\ 0 \\ 1 \end{pmatrix}$$

First, we need to code the “model” the derivative functions given in (7),

```
function dx = rktest2(~,x)

    A = [ -0.6, -3.1, -3.2, -0.5, -1.2
          -0.1,  0.5,  1.3,  0.9, -1.3
          -2.1, -0.9, -2.0, -0.1, -2.8
           0.0, -3.6, -4.4, -1.9, -0.9
          -0.6,  1.5,  1.3,  0.4, -1.3 ];
    b = [ 0; -0.2; -0.1; 0.1; 0];
    dx = A*x + b;

end
```

Since the Runge Kutta program we coded earlier anticipates t as the first argument of the function, we should have put in t in the first line. However, we did not use t in the model, so Matlab suggests using the symbol “~” as a placeholder.⁶

Using this test model and the given initial conditions, the following command lines can be used to run the model from $t = 0$ to $t = 100$ with npts=5000 :

```
>> [tsoln,xsoln]=runge_kutta(@rktest2,0,100,[-4;3;-2;0;1],5000);
>> plot(tsoln,xsoln,'linewidth',2);
>> xlabel('Time (secs)');ylabel('x_i');
>> legend('x_1','x_2','x_3','x_4','x_5')
>> shg
```

This yields the plots given in Figure 1.

⁶ Matlab operations will still proceed even if one had included t instead.

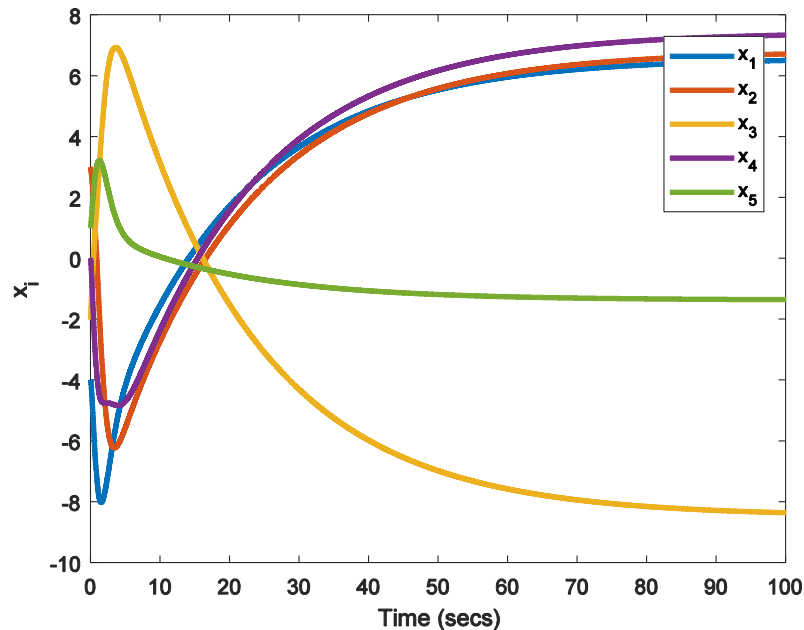


Figure 1. Plot of the numerical solution of (7).

5. Matlab ODE solvers

There are several ODE solvers available in Matlab. One particular solver called “**ODE45**” implements a Runge-Kutta variation known as the Dormand-Prince algorithm. This version simultaneously solves a pair of 4th order and 5th order Runge-Kutta updates. If the difference between these updates is above a prescribed tolerance, the time increment Δt_k will be decreased and the updates are recalculated. Likewise, if the difference is smaller than a satisfactory value, the time increment Δt_k will be increased. This is known as an “error-correcting” implementation, which will then yield a t vector that has variable increments.

The most basic format is given by: **[tsoln, xsoln]=ode45(@fcn, [tinit, tfinal], xinit)**. Thus, for our test example, we could use the following command lines:

```
>> [tsoln,xsoln]=ode45(@rktest2,[0,100],[-4;3;-2;0;1]);
>> plot(tsoln,xsoln,'linewidth',2);
>> xlabel('Time (secs)');ylabel('x_i');
>> legend('x_1','x_2','x_3','x_4','x_5')
>> shg
```

This would yield a very similar plot shown in Figure 1.

There are several capabilities that accompany **ODE45**. We discuss a few of these in Appendix B.

Appendix A. State-space Formulation of High Order ODE

The main approach is to first define a vector of variables, x_1, \dots, x_n , where n is the order of the original ODE. The first variable, x_1 , is set to the original variable, say y . The derivatives of first set of variables from x_k ($k = 1, \dots, n - 1$) are used to define x_{k+1} . This means ($dx_k/dt = d^k y/dt^k$). The derivative of the last variable, x_n , is then algebraically arranged from the original ODE such that (dx_n/dt) is the single term on the left hand side, while the right hand contains terms in which original variable and variable derivatives are substituted by x_1, \dots, x_n .

Suppose we are given a second-order differential equation such as the van der Pol given by

$$\frac{d^2 y}{dt^2} - \mu(1 - y^2) \frac{dy}{dt} + y = 0 \quad (8)$$

With $n = 2$, we define $x_1 = y$ and $x_2 = dx_1/dt$, the original equation can then be rearranged as

$$\begin{aligned} \frac{d^2 y}{dt^2} &= \mu(1 - y^2) \frac{dy}{dt} - y \\ \rightarrow \frac{dx_2}{dt} &= \mu(1 - x_1^2)x_2 - x_1 \end{aligned}$$

Thus, collecting the other derivatives, we have

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= \mu(1 - x_1^2)x_2 - x_1 \end{aligned} \quad (9)$$

which is in the required state-space formulation (1).

Appendix B. Additional Capabilities of Matlab ODE solvers.

There are several solvers available in Matlab. These include the most commonly used solvers: **ode23**, **ode45**, **ode23s** and **ode15s**.⁷ The first two are considered as “explicit non-stiff solvers”, while the latter two are used to solve “stiff” problems. Often, “stiff” problems occur when the system contains extreme range of characteristic times, i.e. some variables changes very fast while others are very slow. Thus, when non-stiff methods are used, they can often appear to “hang”, sometimes due to significant reduction in time increments during the error-correcting process, other times due to the enhanced sensitivity to round-off errors.

In this section we will describe a few of the extended capabilities of Matlab solvers.

1. Adjustable Options

Users can adjust some of the options of the Matlab ODE solvers via passing a structure parameter. Thus, the `odeset` command will create this parameter with the pre-defined fields that are queried inside the Matlab solvers. For example,

```
>> opt1 = odeset
opt1 =
  struct with fields:
    AbsTol: []
    BDF: []
    Events: []
    InitialStep: []
    Jacobian: []
    JConstant: []
    JPattern: []
    Mass: []
    MassSingular: []
    MaxOrder: []
    MaxStep: []
    NonNegative: []
    NormControl: []
    OutputFcn: []
    OutputSel: []
    Refine: []
    RelTol: []
    Stats: []
    Vectorized: []
    MStateDependence: []
    MvPattern: []
    InitialSlope: []
```

⁷ Another solver, **bvp4c**, is available and used to solve boundary value problems. Since it has additional setup and conditions, we have not included with these sets of ODE solvers.

Among these fields, the relevant options for **oder45** are **'AbsTol'**, **'Refine'**, **'RelTol'**, **'MaxStep'** and **'InitialStep'**⁸. These adjust the accuracy of the output by allowing the user to: set the absolute tolerance, increase the refinement of the time increments (e.g. for smoother results in some cases), set the relative tolerance, set the maximum step size or set the initial step size. These fields can be changed directly by either the command:

```
opt1.AbsTol=1e-8, if opt1 has already been defined via odeset,
or
opt1=odeset('AbsTol',1e-8), if opt1 has not yet been defined.
```

Once the option structure parameter has been defined, then it can be passed to the solver as the argument after the initial conditions, i.e.

```
>> opt1=odeset('AbsTol',1e-8);
>> [tSoln,xSoln]=ode45(@rktest2,[0,100],[-4;3;-2;0;1],opt1);
```

2. Passing parameters to the model function.

Sometimes the parameters of a model need to be unspecified inside the model. Instead, they may originate externally and need to be passed on to the model function via the Matlab solver. These parameters could either be an array or a structure variable.

Let us modify the model **rktest2** given in section 4 to allow external specification of matrix **A** and vector **b** via a structure **param** with fields **A** and **b**.

```
function dx = rktest3(~,x,A,b)
    dx = A*x + b;
end
```

First, we need to define matrix **A** and vector **b**. Then we need to pass them through the model function via the entries following the placeholder for the 'options'. If no options are being modified, a null entry is used for options. Thus,

⁸ The fields **'Jacobian'**, **'JConstant'** and **'JPattern'** allow the users to specify how the Jacobians are supplied or used by some solvers. The fields **'Mass'**, **'MassSingular'**, **'MStateDependence'**, **'MvPattern'** and **'InitialSlope'** are used when solving highly stiff and/or differential-algebraic equations (also known as DAEs). The fields **'BDF'**, and **'MaxOrder'** has specific relevance to **oder15s**.


```

>> A=[-0.6000   -3.1000   -3.2000   -0.5000   -1.2000
      -0.1000    0.5000    1.3000    0.9000   -1.3000
      -2.1000   -0.9000   -2.0000   -0.1000   -2.8000
           0   -3.6000   -4.4000   -1.9000   -0.9000
      -0.6000    1.5000    1.3000    0.4000   -1.3000];
>> b=[0;-0.2;-0.1;0.1;0];
>> [tsoln,xsoln]=ode45(@rktest3,[0,100],[-4;3;-2;0;1],[],A,b);
>> plot(tsoln,xsoln,'linewidth',2);
>> xlabel('Time (secs)');ylabel('x_i');
>> legend('x_1','x_2','x_3','x_4','x_5')
>> shg

```

3. Evaluation at specified points of the independent variable

The Matlab solvers will most likely have non-uniform step sizes. Thus, the values at specified points of independent variable, say t , will need to be interpolated. To do so, two steps are needed. First, the output will have to be in the form of a structure, which is recognized by Matlab when only a single output is specified, i.e.

```

>> soln1=ode45(@rktest3,[0,100],[-4;3;-2;0;1],[],A,b);
>> soln1

soln1 =

  struct with fields:

      solver: 'ode45'
  extdata: [1x1 struct]
         x: [1x70 double]
         y: [5x70 double]
       stats: [1x1 struct]
       idata: [1x1 struct]

```

The independent variable array can be accessed as `soln1.x` while the state variable array can be accessed as `soln1.y`. Note however that these results are transposed versions, i.e. they are stored row-wise instead of column-wise.

Suppose we want the values to be specified at 15 evenly spaced points from 0 to 100, then we can use the function: `deval()`. Thus,

```

>> newt=linspace(0,100,15);
>> newx=deval(soln1,newt);

```

This will yield the required interpolated values. For instance, we could use this to include markers as follows:

```
>> plot(soln1.x,soln1.y,'linewidth',2);
>> hold on
>> plot(newt,newx(1,:),'o','linewidth',2,'MarkerFaceColor','w');
>> plot(newt,newx(2,:),'s','linewidth',2,'MarkerFaceColor','w');
>> plot(newt,newx(3,:),'d','linewidth',2,'MarkerFaceColor','w');
>> plot(newt,newx(4,:),'v','linewidth',2,'MarkerFaceColor','w');
>> plot(newt,newx(5,:),'^','linewidth',2,'MarkerFaceColor','w');
>> hold off
>> xlabel('Time (secs)');ylabel('x_i');
>> shg
```

This interpolation capability can be used together with the parameter passing capability discussed in the previous item to obtain estimates of the model, i.e. if the fitness of the model is based on an error index such as sum of squared errors between the predicted values and raw data at the corresponding raw t -data. Thus, the optimum parameters can be determined using optimizers such as **fminsearch** or **fminbnd** to minimize the error index.