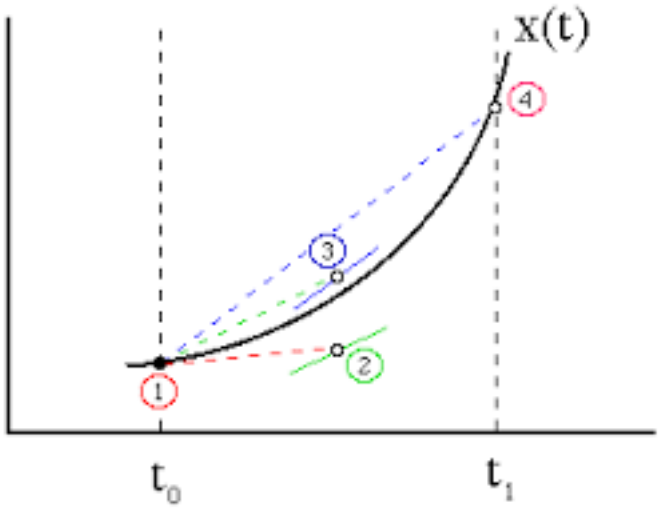
Ordinary Differential Equations: Runge-Kutta methods

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/ode_runge_kutta/ode_runge_kutta.pdf

---



A fourth order Runge Kutta step involves several initial test steps.

ODE's by a Runge-Kutta method

Given an ODE: $y' = f(t, y), y(t_0) = y_0$, consider the Runge-Kutta family of solution methods, which are more accurate than the Euler method.

1 The Runge-Kutta idea

In class, it was explained that, in order to solve an ODE of the form

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

a Runge-Kutta method advances one step at a time. But each step is preceded by a number of trial steps that essentially sample the right hand side function. The result is that the error in a single step (the local truncation error) can be much reduced. This, in turn, means that the overall error, say at a final time of interest, is similarly reduced.

A Runge-Kutta method has an *order*, which reflects the global accuracy. Thus, if we use a Runge-Kutta method of order 2 with a step size of `dt`, we expect that the error in our approximation at any fixed time `T` will tend to decrease like $O(dt^2)$. This is the benefit of using a Runge-Kutta scheme.

The cost of using a Runge-Kutta scheme is reflected in the fact that a scheme of order k typically needs k trial steps (and sometimes more) to produce the next approximate solution.

2 Several Runge Kutta methods

Assume we have already computed y_n, and have chosen a stepsize dt. The computation of the next approximation, y_{n+1} by Runge Kutta method involves computing quantities known as *stages*. Here we will symbolize these quantities by subscripted variables k. Here are examples of Runge-Kutta methods of orders 1, 2, 3 and 4, in which the computation and use of the stages can be examined:

1. The Euler method (order 1):

   $k_1 = dt\, y'(t_n, y_n)$
   $y_{n+1} = y_n + k_1$

2. Heun's method (order 2):

   $k_1 = dt\, y'(t_n, y_n)$
   $k_2 = dt\, y'(t_{n+1}, y_n + k_1)$
   $y_{n+1} = y_n + 0.5\, k_1 + 0.5\, k_2$

3. Strong Stability Preserving Runge Kutta (order 3):

   $k_1 = dt\, y'(t_n, y_n)$
   $k_2 = dt\, y'(t_n + dt, y_n + k_1)$
   $k_3 = dt\, y'(t_n + 0.5\, dt, y_n + 0.25k_1 + 0.25k_2)$
   $y_{n+1} = y_n + \frac{1}{6}(k_1 + k_2 + 4k_3)$

4. Classical Runge Kutta (order 4):

   $k_1 = dt\, y'(t_n, y_n)$
   $k_2 = dt\, y'(t_n + 0.5\, dt, y_n + 0.5\, k_1)$
   $k_3 = dt\, y'(t_n + 0.5\, dt, y_n + 0.5\, k_2)$
   $k_4 = dt\, y'(t_n + dt, y_n + k_3)$
   $y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

3 Butcher table

There is a standard format for Runge Kutta methods. This allows a rule of s stages to be summarized by an $s \times s$ array of numbers known as a Butcher table. The general form is:

c_1	$a_{1,1}$	$a_{1,2}$	...	$a_{1,s}$
c_2	$a_{2,1}$	$a_{2,2}$	...	$a_{2,s}$
...	...	...	...	...
c_s	$a_{s,1}$	$a_{s,2}$	...	$a_{s,s}$
	b_1	b_2	...	b_s

This table is interpreted as follows:

The value of y_{n+1} is computed using the b coefficients and the values of the s stages k_1 through k_s:

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

The value c_i is actually the sum of the values $a_{i,j}$, and is used to determine the value of time at which the derivative $f(t,y)$ is to be evaluated at the i-th stage. Specifically:

$$t = t_i + c_i\, dt$$

The values $a_{i,:}$ are used to compute the y argument of $f(t,y)$ at the i-th stage. Specifically:

$$y_{n+1} = y_n + \sum_{i=1}^{2} a_{i,j}\, k_j$$

This means that the i-th stage is computed by

$$k_i = dt\, f(t,y) = dt\, f(t_i + c_i\, dt, y_n + \sum_{i=1}^{2} a_{i,j}\, k_j)$$

For an explicit Runge-Kutta scheme, the Butcher table is lower triangular.

Thus, the table for the Runge-Kutta scheme of order 4 is

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	0
	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

If you compare the Butcher table to the original description, you can see how the information has been arranged:

$k_1 = dt\, y'(t_n, y_n)$
$k_2 = dt\, y'(t_n + 0.5\, dt, y_n + 0.5\, k_1)$
$k_3 = dt\, y'(t_n + 0.5\, dt, y_n + 0.5\, k_2)$
$k_4 = dt\, y'(t_n + dt, y_n + k_3)$
$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

4 Euler is an rk1 method

Assume we have already computed y_n, and have chosen a stepsize dt. The computation of the next approximation, y_{n+1} by Runge Kutta method involves computing quantities known as *stages*. Here we will symbolize these quantities by subscripted variables k. The Euler method involves just a single stage, k_1. Each iteration looks like this:

$k_1 = dt\, y'(t_n, y_n)$
$y_{n+1} = y_n + k_1$

Because the Euler method is so simple, it's not clear why we create the temporary variable k_1, but as we look at higher order methods, this approach will make sense.

5 Exercise: Write an rk1 code

Start from this pseudocode, and create a MATLAB file called *rk1.m* which implements the Euler method using the Runge-Kutta format:

```
Name: rk1
Input: yprime, n, tspan, y0
Output: t, y

yprime evaluates the right hand side of the ODE.
n is the number of steps to take.
tspan is a vector containing first and last times.
y0 is a vector containing the initial condition.

t is a vector of computed times.
y is a vector of computed ODE solutions.

BEGIN FUNCTION
  Set t to equally spaced values in tspan(1) to tspan(2).
  Set dt to the size of single time step.
  Set the first entry of y to y0.

  LOOP N TIMES ON I
    Set k1 to dt times the derivative at t(i) and y(i)
    Set the next y(i+1) to y(i) plus k1
  END LOOP

END FUNCTION
```

Listing 1: Pseudocode for rk1.m

6 Exercise: Test the rk1 code

Consider the following **expsin** test problem:

$$y'(t) = \cos(t) * y$$
$$y(0) = 1.0$$
$$y_{exact} = e^{\sin(t)}$$

Create a file *expsin.m* which accepts the current values of t and y, and returns the value of the right hand side of the ODE.

Use your `rk1()` code to take 100 steps, from 0.0 to 10.0. Make a plot of the solution (t,y(t)).

How can you judge whether your results are close to the correct solution?

7 Heun's algorithm is an rk2 method

The Heun ODE solver can be written in the following RK form:

$k_1 = dt\, y'(t_n, y_n)$
$k_2 = dt\, y'(t_{n+1}, y_n + k_1)$
$y_{n+1} = y_n + 0.5\, k_1 + 0.5\, k_2$

Thus, the method involves the computation of two stages, k_1 and k_2, which are then combined to form the solution estimate. Heun's algorithm has order 2, so it is more accurate than the Euler method.

8 Exercise: Write and test an rk2 code

Make a copy of your *rk1.m* file and call it *rk2.m*. Make the changes necessary so that it implements Heun's method. This should only involve inserting one new line and modifying another.

```
Name: rk2
Input: yprime, n, tspan, y0
Output: t, y

yprime evaluates the right hand side of the ODE.
n is the number of steps to take.
tspan is a vector containing first and last times.
y0 is a vector containing the initial condition.

t is a vector of computed times.
y is a vector of computed ODE solutions.

BEGIN FUNCTION
  Set t to equally spaced values in tspan(1) to tspan(2).
  Set dt to the size of single time step.
  Set the first entry of y to y0.

  LOOP N TIMES ON I
    Set k1 to dt times the derivative at t(i) and y(i)
    Set k2 to dt times the derivative at t(i)+dt and y(i)+k1.
    Set the next y(i+1) to the previous y plus (k1+k2)/2.
  END LOOP

END FUNCTION
```

Listing 2: Pseudocode for rk2.m

Now repeat the experiment involving the `expsin()` problem, using your `rk2` code, and plot the resulting solution.

9 Comparing rk1 and rk2 gives us an error estimate

We know that one way to get an error estimate is to compute the difference of two approximate solutions of different order. If we treat the higher order result as though it were the exact solution, then the difference gives us an error estimate ... for the lower order result.

Thus, if we compute vectors of approximate solutions using `rk1` and `rk2`, we can compare them after the computation to make an estimated error plot.

Run your `rk1` and `rk2` codes, saving the solution results as `y1` and `y2`. Also compute the exact solution at the times `t` by running `y=expsin_exact(t)`.

Now make a single plot that display three error measures together:

- (t,abs(y1-y2)) in red
- (t,abs(y-y1)), in green
- (t,abs(y-y2)), in blue

Your red line represents our attempt to estimate the error. Does it come close to either the green line (error in low order approximation) or the blue line (error in high order approximation)?

10 Combining rk1 and rk2 gives us an efficient error estimate

The computations in rk1 are actually duplicated in rk2 exactly. In this sense, the rk1 code is "embedded" in the rk2 code. We can make a new `rk12` code which makes this clear, and which allows us to compute the error estimates as we go.

Create a new file `rk12.m` by copying your *rk2.m* file and modifying it to match this pseudocode:

```
Name: rk12
Input: yprime, n, tspan, y0
Output: t, y, e

yprime evaluates the right hand side of the ODE.
n is the number of steps to take.
tspan is a vector containing first and last times.
y0 is a vector containing the initial condition.

t is a vector of computed times.
y is a vector of computed ODE solutions.
e is a vector of error estimates

BEGIN FUNCTION
  Set t to equally spaced values in tspan(1) to tspan(2).
  Set dt to the size of single time step.
  Set the first entry of y to y0.

  LOOP N TIMES ON I
    Set k1 to dt times the derivative at time t(i) and y(i)
    Set y1 to y(i) + k1
    Set k2 to dt times the derivative at time t(i)+dt and y(i)+k1.
    Set y2 to y(i) plus (k1+k2)/2.
    Set y(i+1) to y2
    Set e(i+1) to abs ( y2 - y1 )
  END LOOP

END FUNCTION
```

Listing 3: Pseudocode for rk12.m

The quantity `y1` is the order 1 Euler estimate, while `y2` is the Heun order 2 estimate, and their difference is used as our approximate error.

Make sure that your new code is working properly by using it to rerun the `expsin` test.

To get an idea of how the individual errors can grow, try this command:

```
  [ t, y, e ] = rk12 ( @(t,y)expsin(t,y), n, tspan, y0 );
  plot ( t, y, 'b-', t, cumsum(abs(e)), 'r-' )
```

11 An adaptive ODE method

Now suppose that we want to do an adaptive computation, so that we will not be specifying a number of steps `n`, from which we can determine a fixed value for `dt`.

Instead, the user specifies an initial value for `dt`, and an error tolerance `tol`. Now, after our `rk12` code takes a single step, we compute the error estimate `e` and do the following:

- if `tol * dt < e`, set `dt = dt/2` and recompute the step;
- if `e < 16 * dt`, accept the step, but also set `dt=2*dt` for the next step;
- otherwise, accept the step, and leave `dt` alone.

```
Name: rk12_adapt
Input: yprime, tspan, y0, dt, tol  <--Note: No "n" is input!
Output: t, y, e

yprime evaluates the right hand side of the ODE.
tspan is a vector containing first and last times.
y0 is a vector containing the initial condition.
dt is a suggestion for the first stepsize
tol is a local error tolerance

t is a vector of computed times.
y is a vector of computed ODE solutions.
e is a vector of error estimates.

BEGIN FUNCTION
  Set i to 1
  Set t(1) to tspan(1)
  Set y(1) to y0

  WHILE T(I) < TSPAN(2)

    LOOP FOREVER

      Set k1 to dt times the derivative at time t(i) and y(i)
      Set y1 to y(i) + k1
      Set k2 to dt times the derivative at time t(i)+dt and y(i)+k1.
      Set y2 to y(i) plus (k1+k2)/2.

      Set t(i+1) to t(i) + dt
      Set y(i+1) to y2
      Set e(i+1) to abs ( y2 - y1 )

      if tol * dt < e(i+1)
        dt = dt / 2  (and repeat LOOP FOREVER)
      elseif e(i+1) < tol * dt / 16.0
        dt = dt * 2
        break out of LOOP FOREVER
      else
        break out of LOOP FOREVER
      end

    END LOOP FOREVER

    i = i + 1

  END WHILE

END FUNCTION
```

Listing 4: Pseudocode for rk12_adapt.m

12 Computing Assignment #11

Try to get this adaptive code working. Test it on the `expsin` example, using initial stepsize `dt=0.1` and `tol=0.1`. Report the number of steps taken (that's the length of the `y` vector). Also report the sum of the absolute value of the entries of `e`.

Turn in: your program *hw11.m* by Friday, November 8.