MATH 934 – RUNGE-KUTTA PROJECT

INSTRUCTOR: DR. ADAM LARIOS

Before You Begin

Read all of "Matlab Introduction," emailed to the class, working through each part in Matlab as you go. If you feel the need for some review, it might be good to go through the document again.

Part I: Making your own ODE solver

Now that you are getting familiar with the basics of Matlab, let's try something a little more useful. Let's solve first-order ODE's (Ordinary Differential Equations) with initial values:

$$\begin{cases} \dfrac{d}{dt}y = f(t,y), \\ \quad y(t_0) = y_0. \end{cases}$$

In this section, you will program two O.D.E. solvers and test their accuracy. The first one is already done for you below. **Type this method out yourself**, and see if you can understand each line. When you are ready, try to run the code. Notice that this code is a function, so you can change the inputs and outputs easily. Here, we take a time interval $t_0 \le t \le T$, where with think of t_0 as the initial time, and T as the final time. Let N be the number of points, and calculate the step size h by $h = (T - t_0)/N$.

FILE: `forwardEuler.m`

```
function [y,t] = forwardEuler(f,t0,T,y0,N)
% Solve dy/dt = f(t,y), y(t0)=y0
% for t0 <= t <= T, with N time steps.
% Sample run:
%   [y_approx, t] = forwardEuler(@(t,y) sin(t + y), 0.0, 5.0, 0.2, 30);
%   close all;
%   plot(t,y_approx,'-o');

h = (T - t0)/(N-1); % Calulate and store the step-size
t = linspace(t0,T,N); % A vector to store the time values.

y = zeros(1,N); % Initialize the Y vector.

y(1) = y0; % Start y at the initial value.

for i = 1:(N-1)
  y(i+1) = y(i) + h*f(t(i),y(i)); % Update approximation y at t+h
end
```

CHECK: Are there any orange or red lines on your scroll bar? These are warnings and errors. Hover your mouse over them to see what they are. Clear them up before running your code. If you have a green box at the top, Matlab cannot detect any errors or warnings in your code. This doesn't necessarily mean your code is error-free though! We will develop a more rigorous test below.

Save the above file as "forwardEuler.m" (without the quotes). It must have this exact name, and must be in the your current working directory for Matlab to be able to find it. (In Matlab, "folders" are called "directories.") Use the command `pwd` (print working directory) to check your current directory, `ls` (list) to list what's in it, and `cd <directory name>` to change directory. You can also navigate directories graphically using the "Current Folder" window.

We will now solve the IVP (initial value problem) given by:

$$\begin{cases} \dfrac{dy}{dt} = -ty^2 \\ y(0) = 1 \end{cases}$$

Run the code by entering these commands in the terminal:

```
>> [y_approx,t]= forwardEuler(@(t,y) -t*y^2, 0.0, 5.0, 1.0, 100);
>> plot(t,y_approx,'-o');
```

This will input the function $f(t, y) = -ty^2$, and the values $t_0 = 0$, $y_0 = 1.0$, $T = 5.0$, and use $N = 100$ grid-points. Then it will plot[1] the approximate solution, t vs. $y_{\text{approx}}(t)$. Try playing with the inputs. Can you increase the number of points to get a smoother-looking graph? What happens when you change the initial value, or even the function?

Note that the exact solution is given by $y(t) = 2/(2 + t^2)$ (call the solution, say, `y_exact`), and plot both `y_exact` and `y_approx` on the same graph. Make sure to base these off the same t values! (See the "Matlab Introduction" document if you need help.) Remember to use "`hold on`" to make two plots on one window (see the Matlab Intro worksheet for a detailed example).

Analyzing Error

An numerical approximation is not really any good unless we can say something about *how good* the approximation is. In particular, one feature we want is for our error to decrease as the resolution (i.e., number of grid-points) increases. There are many way of quantifying error. Usually, we want a **global measure of error**, so it is not enough to take, say, the error at the final time; for example, there could have been crazy errors all along the way, and maybe the two solutions just coincidentally line up at the last time. Also, we usually want our "error" to be a single number, since looking at the error at each time can often provide *too much* information, and it is hard to analyze. Let's choose a simple way to measure the global error. For a **fixed resolution** `N`, we can define the error to be:

```
error = max(abs(y_exact - y_approx));
```

In mathematics, this can be written as

$$\text{error} = \max_{i=1\ldots N} |y(t_i) - y_i|$$

where y_i represents the approximate solution at step i, and $y(t)$ is the exact solution at time t (so that $y(t_i)$ is the exact solution at time t_i).

Task 1a: Write a separate code (call it `eulerError.m`) that runs through increasing resolutions, say resolutions $N = 10, 20, 30, \ldots, 1000$, and calculates the (global) error at each resolution. Use the exact solution to the IVP you computed by hand above, and **call the Euler code as a functio**n (which will be in the file `forwardEuler.m`) to compute the approximate solution. This means you will have a loop over the resolutions, probably one line each to compute the exact and approximate solutions, and a line to compute the error. You will have to save the error in a vector for each resolution as you go, so you will need a "counter" or "index" for your error. You will also have to save your resolution values in a vector.

Next, plot the resolution (x-axis) vs. error (y-axis). You should see the error decreasing as the resolution increases. You can get a better picture still by using a log-log plot. Let's explore that idea briefly.

[1] For more plotting options, type `help plot`.

Log-log plots. Suppose you have some data that seems to approximately fit a $y = cx^p$ pattern for some constant $c > 0$ and some power p. How can we determine the power from just looking at the data? Notice that (so long as $x > 0$ and $y > 0$),

$$\begin{aligned} & y = cx^p \\ \Rightarrow & \log(y) = \log(cx^p) \\ \Rightarrow & \log(y) = \log(c) + p\log(x) \end{aligned}$$

Setting $X = \log(x)$, $Y = \log(y)$, and $C = \log(c)$ we see that

$$Y = pX + C$$

Thus, the slope between $\log(x)$ and $\log(y)$ is p, which is exactly the power. Working *backwards*, if we ever have some data, and we suspect there is a power-relationship, we can figure out the power by looking at a log-log plot and looking at the slope!

Task 1b: Look at a log-log plot of your error. You can do this in Matlab like this:

```
>> close all;
>> plot(log(resolution),log(error));
```

where `resolution` is a vector holding all your resolution values, and `error` is a vector holding all the corresponding errors. Matlab actually has a built-in log-log plotter which makes the axes look better. It works like this:

```
>> close all;
>> loglog(resolution,error);
```

You should see a slope of -1, at least as N gets large. If you see any other slope, there is an error either in your `forwardEuler.m` code, or your error checker. This slope means that

$$\text{global error} \approx CN^{-1} \approx Ch$$

for some constant $C > 0$. Thus, **if you *double* your resolution, error should be cut in *half*.**

Higher-Order Methods. Forward Euler is one of the simplest numerical algorithms for solving an IVP, but its solutions can have a lot of error. Of course, we could decrease the error by using more grid points (or so we hope), but this means we have to do a lot more work (and accumulate more round-off error). For Forward Euler, if we double the resolution (i.e., the number of grid points), our error is only 1/2 of what is was. Is it possible to have a method that when we double the resolution, the error is, say, 1/4 of what is was? That is, could we have a method such that

$$\text{global error} \approx CN^{-2} \approx Ch^2$$

Such a method would be called a *second-order* method. Improved Euler is a second order method, but we also will study a forth-order method (error $\approx CN^{-4} \approx Ch^4$). This means if you double the resolution, the error is 1/16 of what is was! Let us look at a method which turns out to be a second-order method. It has several names, including "Improved Euler," "Runge-Kutta-2 (RK-2)," "Heun's Method" and "Ralston's Method." To understand where it comes from, consider the Euler methods:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) \qquad \text{(Forward Euler)}$$
$$y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1}) \qquad \text{(Backward Euler)}$$

Backward Euler has better stability properties (we will discuss this later), but y_{n+1} is only *implicitly* defined, which means we have to solve an algebraic problem every time step to find y_{n+1} (unless f is very nice, e.g., it is linear, so we can solve it by hand).

We would like to use Backward Euler, but the y_{n+1} on the right-hand side is not known. Instead, we can *approximate* it using forward Euler. We then average the results of the two methods. It looks something like this (each step is carried out sequentially, starting at the top):

$$\begin{cases} y^*_{n+1} = y_n + h \cdot f(t_n, y_n) & \text{(prediction with forward Euler)} \\ y^{**}_{n+1} = y_n + h \cdot f(t_{n+1}, y^*_{n+1}) & \text{(use prediction in backward Euler)} \\ y_{n+1} = \frac{1}{2}(y^*_{n+1} + y^{**}_{n+1}) & \text{(average the predictions)} \end{cases}$$

The final y_{n+1} is what we use as our approximated value. Notice that this is a fully explicit method! It looks a little messy with all the $*$'s, so let's make it a little cleaner by first noting that

$$\begin{aligned} \tfrac{1}{2}(y^*_{n+1} + y^{**}_{n+1}) &= \tfrac{1}{2}[(y_n + h \cdot f(t_n, y_n)) + (y_n + h \cdot f(t_{n+1}, y^*_{n+1}))] \\ \Rightarrow \qquad y_{n+1} &= y_n + \tfrac{h}{2}(f(t_n, y_n) + f(t_n + h, y_n + h \cdot f(t_n, y_n))). \end{aligned} \tag{1}$$

since $t_{n+1} = t_n + h$. Next, note that we are being inefficient, since we compute $f(t_n, y_n)$ multiple times. Therefore, we can just save it as a value, say, k_1, and use it when we need it. We can now write the method like this:

$$\text{(RK-2)} \quad \begin{cases} k_1 = f(t_n, y_n) \\ k_2 = f(t_n + h, y_n + h \cdot k_1) \\ y_{n+1} = y_n + \frac{h}{2} \cdot (k_1 + k_2) \end{cases}$$

This is the Improved Euler Method (or Heun, or RK-2, or Ralphson, etc.). It is an **explicit** method of **order 2**, meaning its error behaves like $C \cdot h^2$ when h is small, where C is some fixed number depending on the ODE problem, but not depending on h. For short-hand, we say it is an $\mathcal{O}(h^2)$ method, using the "Big-O" notation.

Task 2: Prove mathematically that Runge-Kutta-2 (RK-2) is an $\mathcal{O}(h^2)$ method.
Hint: This is easiest to do from equation (1). Follow the steps we did in class, for showing the local truncation error for Forward Euler is bounded by Ch^2, to show that the local truncation for RK-2 is Ch^3. This involves taking derivatives of the equation several times, using the chain rule, using the third-order Taylor Remainder Theorem, and using the equation to replace values as you go. Use this to show the global error is bounded by Ch^2.

Runge-Kutta-4. One of the most widely-used numerical algorithms is the Runge-Kutta-4 (RK-4) method. It is a fourth-order explicit method, given by one of the following equivalent forms.

$$\text{(RK-4)} \quad \begin{cases} k_1 = f(t_n, y_n) \\ k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_1) \\ k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_2) \\ k_4 = f(t_n + h, y_n + h \cdot k_3) \\ y_{n+1} = y_n + \frac{h}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \end{cases} \quad \text{or} \quad \begin{cases} k_1 = h \cdot f(t_n, y_n) \\ k_2 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{1}{2} \cdot k_1) \\ k_3 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{1}{2} \cdot k_2) \\ k_4 = h \cdot f(t_n + h, y_n + k_3) \\ y_{n+1} = y_n + \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \end{cases}$$

Task 3a: Adapt your `forwardEuler.m` code to a new code called `rk4.m`, which solves the IVP using Runge-Kutta-4. It will be essentially the same, but add a few more lines.

Task 3b: Test your implementation by computing the resolution vs. error as in Task 1b. The slope in the log-log plot should now be -4 instead of -1. When you see that slope, you will know that your code is working, and you will have a fourth-order IVP solver!

When you have finished tasks 1-3, show them to me, and I will check them off. If you get stuck, come by, and we can work though it together! =)

Have fun, and happy coding!

More Information on Runge-Kutta Methods

In general, one can have Runge-Kutta methods of any order. An p-stage method can be give by

$$
\text{(RK-p)} \quad \begin{cases}
\quad k_1 = f(t_n, y_n) \\
\quad k_2 = f(t_n + \alpha_2 h, y_n + \beta_{21} h \cdot k_1) \\
\quad k_3 = f(t_n + \alpha_3 h, y_n + \beta_{31} h \cdot k_1 + \beta_{32} h \cdot k_2) \\
\quad\ \vdots \quad \vdots \\
\quad k_p = f(t_n + \alpha_p h, y_n + \beta_{p1} h \cdot k_1 + \beta_{p2} h \cdot k_2 + \cdots + \beta_{p,p-1} h \cdot k_{p-1}) \\
y_{n+1} = y_n + h(c_1 k_1 + c_2 k_2 + \cdots + c_p k_p)
\end{cases}
$$

For any given method the constants α_i, β_i, and c_i are usually looked up in a table (they are determined by working out the local truncation error with Taylor series, and choosing the constants to make all the terms cancel up to a desired order). They are typically given in the form of a "**Butcher-tableau**", named after the New Zealand mathematician John Butcher, who works at the University of Auckland. For the method to be *consistent* (i.e., for the local truncation error $\tau \to 0$ as $h \to 0$), it is sufficient for $\sum_{j=1}^{p} \beta_{i,j} = \alpha_i$ for each $i = 2, 3, \ldots, p$. For example, (RK-p) is given by the Butcher tableau:

$$
\begin{array}{c|ccccc}
0 & & & & & \\
\alpha_2 & \beta_{21} & & & & \\
\alpha_3 & \beta_{31} & \beta_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
\alpha_p & \beta_{p1} & \beta_{p2} & \cdots & \beta_{p,p-1} & \\
\hline
 & c_1 & c_2 & \cdots & c_{p-1} & c_p
\end{array}
$$

Forward Euler (RK-1) is given by the Butcher tableau:

$$
\begin{array}{c|c}
0 & \\
\hline
 & 1
\end{array}
$$

Improved Euler (RK-2) is given by the Butcher tableau:

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

And RK-4 is given by the Butcher tableau:

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

Caution! A p-stage might not be an order-p method! For example, a 5$^{\text{th}}$-order method requires *6 stages.*

Matlab's ODE solver `ode45.m` is based on Erwin Fehlberg's method, which is two methods combined into one, allowing for an adaptive step-size. They have the same coefficients α_i, β_i, and only differ in the c_i coefficients, so we can write them in the same table as:

0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	-7200/2197	7296/2197			
1	439/216	-8	3680/513	-845/4104		
1/2	-8/27	2	-3544/2565	1859/4104	-11/40	
	16/135	0	6656/12825	28561/56430	-9/50	2/55
	25/216	0	1408/2565	2197/4104	-1/5	0

The first bottom row is used to compute a 4th-order accurate solution. The second bottom row is used to compute a 5th-order accurate solution. If the two results are significantly different, the step size h is decreased (often in some optimal way), and the calculation is repeated for that step.