MATH2071: LAB 2: Explicit ODE methods

1 Introduction

In this lab we consider solution methods for ordinary differential equations (ODEs). We will be looking at two classes of methods that excel when the equations are smooth and derivatives are not too large. This lab will take two class sessions. If you print this lab, you may prefer to use the pdf version.

The lab begins with an introduction to Euler's (explicit) method for ODEs. Euler's method is the simplest approach to computing a numerical solution of an initial value problem. However, it has about the lowest possible accuracy. If we wish to compute very accurate solutions, or solutions that are accurate over a long interval, then Euler's method requires a large number of small steps. Since most of our problems seem to be computed "instantly," you may not realize what a problem this can become when solving a "real" differential equation.

Applications of ODEs are divided between ones with space as the independent variable and ones with time as the independent variable. We will use x as independent variable consistently. Sometimes it will be interpreted as a space variable (x-axis) and sometimes as time.

A number of methods have been developed in the effort to get solutions that are more accurate, less expensive, or more resistant to instabilities in the problem data. Typically, these methods belong to "families" of increasing order of accuracy, with Euler's method (or some relative) often being the member of the lowest order.

In this lab, we will look at "explicit" methods, that is, methods defined by an explicit formula for y_{k+1}, the approximate solution at the next time step, in terms of quantities derivable from previous time step data. In a later lab, we will address "implicit" methods that require the solution of an equation in order to find y_{k+1}. We will consider the Runge-Kutta and the Adams-Bashforth families of methods. We will talk about some of the problems of implementing the higher order versions of these methods. We will try to compare the accuracy of different methods applied to the same problem, and using the same number of steps.

Runge-Kutta methods are "single-step" methods while Adams-Bashforth methods are "multistep" methods. Multistep methods require information from several preceding steps in order to find y_{k+1} and are a little more difficult to use. Nonetheless, both single and multistep methods have been very successful and there are very reliable Matlab routines (and libraries for other languages) available to solve ODEs using both types of methods.

2 Matlab hint

Matlab vectors can be either row vectors or column vectors. Unlike ordinary vectors in theoretical work, row and column vectors behave almost as if they belong to different vector spaces. They cannot, for example be added together and a matrix can only be multiplied on the right by a column vector or on the left by a row vector. The reason for the distinction is that vectors are really special cases of matrices with a "1" for the other dimension. A row vector of length n is really a $1 \times n$ matrix and a column vector of length n is really a $n \times 1$ matrix.

You should recall that row vectors are separated by commas when using square brackets to construct them and column vectors are separated by semicolons. It is sometimes convenient to write a column vector as a column. In the following expressions

```
rv = [ 0, 1, 2, 3];
cv = [ 0; 1; 2; 3; 4];
cv1 = [0
       1
       2
       3
       4];
```

The vector `rv` is a row vector and the vectors `cv` and `cv1` are column vectors.

If you do not tell Matlab otherwise, Matlab will generate a row vector when it generates a vector. The output from `linspace` is, for example, a row vector. Similarly, the following code

```
for j=1:10
  rv(j)=j^2;
end
```

results in the vector `rv` being a row vector. If you wish to generate a column vector using a loop, you can either first fill it in with zeros

```
cv=zeros(10,1);
for j=1:10
  cv(j)=j^2;
end
```

or use two-dimensional matrix notation

```
for j=1:10
  cv(j,1)=j^2;
end
```

3 Euler's method

A very simple ordinary differential equation (ODE) is the *explicit scalar first-order initial value problem*:

$$\begin{aligned} \frac{dy}{dx} &= f_{\text{ode}}(x,y) \\ y(x_0) &= y_0. \end{aligned}$$

The equation is *explicit* because dy/dx can be written explicitly as a function of x and y. It is *scalar* because we assume that $y(x)$ is a scalar quantity, not a vector. It is *first-order* because the highest derivative that appears is the first derivative dy/dx. It is an *initial value problem* (IVP) because we are given the value of the solution at some time or location x_0 and are asked to produce a formula for the solution at later times.

An *analytic solution* of an ODE is a formula $y(x)$, that we can evaluate, differentiate, or analyze in any way we want. Analytic solutions can only be determined for a small class of ODE's. The term "analytic" used here is not quite the same as an analytic function in complex analysis.

A "numerical solution" of an ODE is simply a table of abscissæ and approximate values (x_k, y_k) that approximate the value of an analytic solution. This table is usually accompanied by some rule for interpolating solution values between the abscissæ. With rare exceptions, a numerical solution is *always wrong* because there is always some difference between the tabulated values and the true solution. The important question is, how wrong is it? One way to pose this question is to determine how close the computed values (x_k, y_k) are to the analytic solution, which we might write as $(x_k, y(x_k))$.

The simplest method for producing a numerical solution of an ODE is known as *Euler's explicit method*, or the *forward Euler method.* Given a solution value (x_k, y_k), we estimate the solution at the next abscissa by:

$$y_{k+1} = y_k + hy'(x_k, y_k).$$

(The step size is denoted h here. Sometimes it is denoted dx.) We can take as many steps as we want with this method, using the result from one step as the starting point for the next step.

Matlab note: In the following function, the name of the function that evaluates dy/dx is arbitrary. Recall that if you do not know the actual name of a function, but it is contained in a Matlab variable (I often use the variable name "`f_ode`") then you can evaluate the function using the Matlab function using the usual function syntax. Supposing you have a Matlab function m-file named `my_ode.m` and its signature line looks like the following.

```
function fValue=my_ode(x,y)
```

Suppose you wish to call it from inside another function whose signature line looks like the following.

```
function [ x, y ] = ode_solver ( f_ode, xRange, yInitial, numSteps )
```

When you call `ode_solver` using a command such as

```
[ x, y] = ode_solver(@my_ode,[0,1],0,10)
```

then, inside `ode_solver` you can use syntax such as

```
fValue=f_ode(x,y)
```

to call `my_ode`. This is the approach taken in this and future labs.

Matlab has an alternative, slightly more complicated, way to do the same thing. Inside `ode_solver` you can use the Matlab `feval` utility

```
fValue=feval(f_ode,x,y)
```

to call `my_ode`.

Typically, Euler's method will be applied to systems of ODEs rather than a single ODE. This is because higher order ODEs can be written as systems of first order ODEs. The following Matlab function m-file implements Euler's method for a system of ODEs.

```
function [ x, y ] = forward_euler ( f_ode, xRange, yInitial, numSteps )
% [ x, y ] = forward_euler ( f_ode, xRange, yInitial, numSteps ) uses
% Euler's explicit method to solve a system of first-order ODEs
% dy/dx=f_ode(x,y).
% f = function handle for a function with signature
%    fValue = f_ode(x,y)
% where fValue is a column vector
% xRange = [x1,x2] where the solution is sought on x1<=x<=x2
% yInitial = column vector of initial values for y at x1
% numSteps = number of equally-sized steps to take from x1 to x2
% x = row vector of values of x
% y = matrix whose k-th column is the approximate solution at x(k).

x(1) = xRange(1);
h = ( xRange(2) - xRange(1) ) / numSteps;
```

```
y(:,1) = yInitial;
for k = 1 : numSteps
  x(1,k+1) = x(1,k) + h;
  y(:,k+1) = y(:,k) + h * f_ode( x(k), y(:,k) );
end
```

In the above code, the initial value (`yInitial`) is a *column* vector, and the function represented by `f` returns a *column* vector. The values are returned in the *columns* of the matrix `y`, one column for each value of x. The vector `x` is a row vector.

In the following exercise, you will use `forward_euler.m` to find the solution of the initial value problem

$$\begin{aligned} \frac{dy}{dx} &= -y - 3x \\ y(0) &= 1 \end{aligned} \tag{1}$$

The exact analytic solution of this IVP is $y = -2e^{-x} - 3x + 3$.

Exercise 1:

(a) If you have not done so already, copy (use cut-and-paste) the above code into a file named `forward_euler.m`.

(b) Copy the following code into a Matlab m-file called `expm_ode.m`.

```
function fValue = expm_ode ( x, y )
% fValue = expm_ode ( x, y ) is the right side function for
% the ODE dy/dx=-y+3*x
% x is the independent variable
% y is the dependent variable
% fValue represents dy/dx

fValue = -y-3*x;
```

(c) Now you can use Euler's method to march from `y=yInit` at `x=0`:

```
yInit = 1.0;
[ x, y ] = forward_euler ( @expm_ode, [ 0.0, 2.0 ], yInit, numSteps );
```

for each of the values of `numSteps` in the table below. Use at least four significant figures when you record your numbers (you may need to use the command `format short e`), and you can use the first line as a check on the correctness of the code. In addition, compute the error as the difference between your approximate solution and the exact solution at `x=2`, `y=-2*exp(-2)-3`, and compute the ratios of the error for each value of `nstep` divided by the error for the succeeding value of `nstep`. As the number of steps increases, your errors should become smaller and the ratios should tend to a limit.

		Euler's explicit method		
numSteps	Stepsize	Euler	Error	Ratio
10	0.2	-3.21474836	5.5922e-02	_________
20	0.1	__________	__________	_________
40	0.05	__________	__________	_________
80	0.025	__________	__________	_________
160	0.0125	__________	__________	_________
320	0.00625	__________	__________	

Hint: Recall that Matlab has a special index, `end`, that always refers to the last index. Thus, `y(end)` is a short way to write `y(numel(y))` when `y` is either a row vector or a column vector.

(d) You know the error is $O(h^p)$ for some p. There is a simple way to estimate the value of p by successively halving h. If the error were *exactly* Ch^p, then by solving twice, once using h and the second time using $h/2$ and taking the ratio of the errors, you would get

$$\frac{\text{error}(h)}{\text{error}(h/2)} = \frac{Ch^p}{C(h/2)^p} = 2^p.$$

Since the error is only $O(h^p)$, the ratio is only approximately 2^p.

Based on the ratios in the table, estimate the order of accuracy of the method, *i.e.,* estimate the exponent p in the error estimate Ch^p, where h is the step size. p is an integer in this case.

4 The Euler Halfstep (RK2) Method

The "Euler halfstep" or "RK2" method is a variation of Euler's method. It is the second-simplest of a family of methods called "Runge-Kutta" methods. As part of each step of the method, an auxiliary solution, one that we don't really care about, is computed halfway, using Euler's method:

$$\begin{aligned} x_a &= x_k + h/2 \\ y_a &= y_k + 0.5hf_{\text{ode}}(x_k, y_k) \end{aligned} \tag{2}$$

The derivative function is evaluated at this point, and used to take a full step from the original point:

$$\begin{aligned} x_{k+1} &= x_k + h; \\ y_{k+1} &= y_k + hf_{\text{ode}}(x_a, y_a) \end{aligned} \tag{3}$$

Although this method uses Euler's method, it ends up having a higher order of convergence. Loosely speaking, the initial half-step provides additional information: an estimate of the derivative in the middle of the next step. This estimate is presumably a better estimate of the overall derivative than the value at the left end point. The per-step error is $O(h^3)$ and, since there are $O(1/h)$ steps to reach the end of the range, $O(h^2)$ overall. Keep in mind that we do not regard the auxiliary points as being part of the solution. We throw them away, and make no claim about their accuracy. It is only the whole-step points that we want.

In the following exercise we compare the results of RK2 with Euler.

Exercise 2:

(a) Write a Matlab function m-file named `rk2.m` that implements the Euler halfstep (RK2) method sketched above in Equations (2) and (3). Keep the same calling parameters and results as for `forward_euler.m` above. Keeping these the same will make it easy to compare different methods. The following model for the file is based on the `forward_euler.m` file with the addition of the variables `xa` and `ya` representing the auxiliary variables x_a and y_a in Equation (2). Add comments to this outline, including explanations of all the variables in the signature line, and fill in expressions where `???` have been left.

```
function [ x, y ] = rk2 ( f_ode, xRange, yInitial, numSteps )
% [ x, y ] = rk2 ( f_ode, xRange, yInitial, numSteps )
% comments including the signature, meanings of variables,
% math methods, your name and the date

x(1,1) = xRange(1);
h = ( xRange(2) - xRange(1) ) / numSteps;
```

```
y(:,1) = yInitial;
for k = 1 : numSteps
  xa = ??? ;
  ya = ??? ;
  x(1,k+1) = x(1,k) + h;
  y(:,k+1) = y(:,k) + h * f_ode( ??? );
end
```

(b) Use this file to compute the numerical solution of the model ODE for the exponential, `expm_ode.m`, from Exercise 1, from `x = 0.0` to `x = 2.0`, and with the same initial value as in Exercise 1, but using Euler's halfstep method, RK2, with stepsizes below. For each case, record the value of the numerical solution at `x = 2.0`; the error, that is, the difference between the numerical solution and the true solution at the end point `x=2` (`y=-2*exp(-2)-3`); and, the ratios of the error for each value of `numSteps` divided by the error for the succeeding value of `numSteps`. Use at least 4 significant digits when you record values.

numSteps	Stepsize	RK2	RK2 Error	Ratio
10	0.2	-3.274896063	4.2255e-3	__________
20	0.1	__________	__________	__________
40	0.05	__________	__________	__________
80	0.025	__________	__________	__________
160	0.0125	__________	__________	__________
320	0.00625	__________	__________	

(c) Based on the ratios in the table, estimate the order of accuracy of the method, *i.e.,* estimate the exponent p in the error estimate Ch^p. p is an integer in this case.

(d) Compare errors from Euler's method (Exercise 1) and Euler's halfstep method for this problem. You should clearly see that Euler's halfstep method (RK2) converges much faster than Euler's method.

numSteps	Stepsize	Euler Error	RK2 Error
10	0.2	__________	__________
20	0.1	__________	__________
40	0.05	__________	__________
80	0.025	__________	__________
160	0.0125	__________	__________
320	0.00625	__________	__________

(e) Based on the above table, roughly how many steps does Euler require to achieve the accuracy that RK2 has for `numSteps=10`?

(f) You have already found the the error for Euler's method is approximately $C_{\mathrm{E}}h^{p_{\mathrm{E}}}$ and the error for RK2 is approximately $C_{\mathrm{RK2}}h^{p_{\mathrm{RK2}}}$. Based on one or both of these estimates, roughly how many steps would Euler require to achieve the accuracy that RK2 has for `numSteps=320`? Explain your reasoning.

(g) Check that the accuracy obtained using Euler's method with your estimated number of steps is comparable to the accuracy that RK2 has for `numSteps=320`.

Be sure to include a copy of your `rk2.m` file when you send me your summary.

5 Runge-Kutta Methods

The idea in Euler's halfstep method is to "sample the water" between where we are and where we are going. This gives us a much better idea of what f is doing, and where our new value of y ought to be. Euler's method ("RK1") and Euler's halfstep method ("RK2") are the junior members of a family of ODE solving methods known as "Runge-Kutta" methods.

To develop a higher order Runge-Kutta method, we sample the derivative function f at even more "auxiliary points" between our last computed solution and the next one. These points are *not* considered part of the solution curve; they are just a computational aid. The formulas tend to get complicated, but let me describe the next one, at least.

The third order Runge Kutta method "RK3," given x, y and a stepsize h, computes two intermediate points:

$$\begin{aligned} x_a &= x_k + .5h \\ y_a &= y_k + .5h f_{\text{ode}}(x_k, y_k) \\ x_b &= x_k + h \\ y_b &= y_k + h(2f_{\text{ode}}(x_a, y_a) - f_{\text{ode}}(x_k, y_k)) \end{aligned} \tag{4}$$

and then estimates the solution as:

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= y_k + h(f_{\text{ode}}(x_k, y_k) + 4.0 f_{\text{ode}}(x_a, y_a) + f_{\text{ode}}(x_b, y_b))/6.0 \end{aligned} \tag{5}$$

The global accuracy of this method is $O(h^3)$, and so we say it has "order" 3. Higher order Runge-Kutta methods have been computed, with those of order 4 and 5 the most popular.

Exercise 3:

(a) Write a Matlab function m-file called `rk3.m` with the signature

```
function [ x, y ] = rk3 ( f_ode, xRange, yInitial, numSteps )
% comments including the signature, meanings of variables,
% math methods, your name and the date
```

that implements the above algorithm. You can use `rk2.m` as a model.

(b) Repeat the numerical experiment in Exercise 2 (using `expm_ode`) and fill in the following table. Use the first line of the table to confirm that you have written the code correctly.

			RK3	
numSteps	Stepsize	RK3	Error	Ratio
10	0.2	-3.27045877	2.1179e-04	__________
20	0.1	__________	__________	__________
40	0.05	__________	__________	__________
80	0.025	__________	__________	__________
160	0.0125	__________	__________	__________
320	0.00625	__________	__________	

(c) Based on the ratios in the table, estimate the order of accuracy of the method, *i.e.,* estimate the exponent p in the error estimate Ch^p. p is an integer in this case.

(d) Compare errors from the RK2 method (Exercise 2) and the RK3 method for this problem. You should clearly see that RK3 converges much faster than RK2.

numSteps	Stepsize	RK2 Error	RK3 Error
10	0.2	----------	----------
20	0.1	----------	----------
40	0.05	----------	----------
80	0.025	----------	----------
160	0.0125	----------	----------
320	0.00625	----------	----------

(e) Based on the above table, roughly how many steps does RK2 require to achieve the accuracy that RK3 has for `numSteps=10`?

(f) You have already found the the error for RK2 is approximately $C_{\text{RK2}}h^{p_{\text{RK2}}}$ and the error for RK3 is approximately $C_{\text{RK3}}h^{p_{\text{RK3}}}$. Based on one or both of these estimates, roughly how many steps would RK2 require to achieve the accuracy that RK3 has for `numSteps=320`? Explain your reasoning.

(g) Check that the accuracy obtained using RK2 with your estimated number of steps is comparable to the accuracy that RK3 has for `numSteps=320`.

Exercise 4: You have not tested your code for *systems* of equations yet. In this exercise you will do so by solving the "system"

$$\begin{aligned}\frac{dy_1}{dx} &= -y_1 - 3x \\ \frac{dy_2}{dx} &= -y_2 - 3x. \end{aligned} \tag{6}$$

You can see that this "system" is really (1) twice, so you can check y_1 and y_2 against each other and against your earlier work.

(a) It turns out that the file `expm_ode.m` will return a vector if it is given a vector for `y`. To see this fact in action, the following command:

```
fValue = expm_ode(1.0,[5;6])
```

Please include the value of `fValue` in your summary file. If `fValue` is not a *column vector* of length 2, you have a mistake somewhere. Fix it before continuing. (Hint: First make sure your vector `[5;6]` has a semicolon in it and is a column vector.)

(b) Solve the system (6) using `rk3` and `expm_ode` on the interval [0,2] starting from the initial value *vector* [5;6], with 40 steps, Call the solution `ysystem`. What is `ysystem(:,end)`?

(c) Solving the system (6) amounts to solving (1) twice, once with initial value y(0)=5, and once more with initial value y(0)=6. Solve the scalar IVP (1) twice using `rk3` and `expm_ode` on the interval [0,2] using 40 steps, once with initial value `5` and once with initial value `6`. Call the solutions `y1` and `y2`. What are `y1(end)` and `y2(end)`? If these values are different from those of `ysystem(:,end)`, you have a mistake somewhere. Fix your mistake before continuing.
Debugging hint: Check if `y1(1)` and `ysystem(1,1)` or `y2(1)` and `ysystem(2,1)` also disagree. Use `format long` so you can see all decimal places. One or both of these probably disagree. Using the debugger, look at `xa`, `ya`, `xb`, `yb`, `x(2)` and `y(:,2)` and compare them with the respective results from `expm_ode`. The earliest difference you see comes from an incorrect line of code.

(d) Use the following code to compare the solutions for all x values.

```
norm(ysystem(1,:)-y1)/norm(y1)  % should be roundoff or zero
norm(ysystem(2,:)-y2)/norm(y2)  % should be roundoff or zero
```

Exercise 5:

The equation describing the motion of a pendulum can be described by the single dependent variable θ representing the angle the pendulum makes with the vertical. The coefficients of the equation depend on the length of the pendulum, the mass of the bob, and the gravitational constant. Assuming a coefficient value of 3, the equation is

$$\frac{d^2\theta}{dx^2} + 3\sin\theta = 0$$

and one possible set of initial conditions is

$$\begin{aligned} \theta(x_0) &= 1 \\ \frac{d\theta}{dx}(x_0) &= 0 \end{aligned}$$

This second order equation can be written as a system

$$\begin{aligned} \frac{dy}{dx} &= \begin{pmatrix} y_2 \\ -3\sin y_1 \end{pmatrix} \\ y(0) &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \end{aligned}$$

(Recall that this transformation is accomplished by the change of variables $y_1 = \theta$ and $y_2 = d\theta/dx$.)

(a) Write a function m-file named `pendulum_ode.m` with signature

```
function fValue = pendulum_ode(x,y)
% fValue = pendulum_ode(x,y)
% comments including meanings of variables,
% math methods, your name and the date
```

Be sure to put comments after the signature line and wherever else they are needed. Be sure that you return a *column* vector.

(b) Compare the solutions using the first order Euler method, and the third order RK3 method, using 1000 steps for each method, over the interval 0 to 25, with initial condition `y=[1;0]`. Fill in the following table, where `n` is the value of the subscript for `x(n)=6.25`, *etc.*, within roundoff.

x	n	Euler		RK3	
0.00	1	1.00	0.00	1.00	0.00
6.25	___	__________	__________	__________	__________
12.50	___	__________	__________	__________	__________
18.75	___	__________	__________	__________	__________
25.00	___	__________	__________	__________	__________

Hint: You may want to use the Matlab `find` function. Use "`help find`" or the help menu for details.

(c) Note that conservation of energy guarantees that θ should stay between -1 and 1. Generate plots of θ *vs.* x for the forward Euler and Runge-Kutta-3 solutions. Can you see anything in the plot of the Euler's method solution that might indicate that it is wrong, aside from conservation of energy? You do not have to send me a copy of the plot.

(d) Solve the ODE using Euler's method again, but use 10,000 steps instead of 1000. Plot both θ from the refined Euler solution and θ from the original RK3 solution on the same plot and include it with your summary. Your plots should illustrate the fact that refining the mesh helps Euler, but it is still inaccurate compared with RK3. Please include this plot with your summary.

Message: Euler's method is not very accurate, even on nice problems. Our solution curve from Euler's method looks smooth, but it's fundamentally flawed! You must not accept a solution just because your formulæ seem to be correctly copied and the solution "looks nice."

6 Stability

You have seen a considerable amount of theory about stability of methods for ODEs, and you will see more in the future. Explicit methods generally are conditionally stable, and require that the step size be smaller than some critical value in order to converge. It is interesting to see what happens when this stability limit is approached and exceeded. In the following exercise you will drive Euler's method and Runge-Kutta third-order methods unstable using the `expm_ode` function from before. You should observe very similar behavior of the two methods. This behavior, in fact, is similar to that of most conditionally stable methods.

> **Exercise 6**: Each part of this exercise results in a plot. Please include each of the plots with your summary.
>
> (a) Use `forward_euler` to solve the ODE using `expm_ode`, over the interval [0,20], starting from `y=20` and using `numSteps=40`. This solution is well within the stable regime. Plot this case. What is the the step size?
>
> (b) Now solve the same IVP using `numSteps=30, 20, 15, 12` and `10`. The consequence of decreasing the numbers of intervals is to increase the step size. Plot these several solutions on the same plot. You should observe increasing "plus-minus" oscillations in the solutions. What are the step sizes of each of these cases?
>
> (c) Same, but using `numSteps=8`. This solution "explodes" in a "plus-minus" fashion. Plot this case. What is the step size of this case?
>
> (d) Use `rk3` to solve the ODE using `expm_ode`, over the interval [0,20], starting from `y=20` and using `numSteps=20, 10, 9` and `8` steps. Plot these on the same plot. The first of these is stable. What are the step sizes of these four cases?
>
> (e) Same, but with `numSteps=7`. This solution "explodes" in a "plus-minus" fashion. Plot it. What is the step size of this case?

The message you should get from the previous exercise is that you can observe poor solution behavior when you are near the stability boundary for any method, no matter what the theoretical accuracy is. The "poor behavior" appears as a "plus-minus" oscillation that can shrink, grow, or remain of constant amplitude (unlikely, but possible). It can be tempting to accept solutions with small "plus-minus" oscillations that die out, but it is dangerous, especially in nonlinear problems, where the oscillations can cause the solution to move to a nearby curve with different initial conditions that has very different qualitative behavior from the desired solution.

7 Adams-Bashforth Methods

Like Runge-Kutta methods, Adams-Bashforth methods want to estimate the behavior of the solution curve, but instead of evaluating the derivative function at new points close to the next solution value, they look at the derivative at old solution values and use interpolation ideas, along with the current solution and derivative, to estimate the new solution. This way they don't compute solutions at auxiliary points and then throw the auxiliary values away. The savings can result in increased efficiency.

Looked at in this way, the forward Euler method is the first order Adams-Bashforth method, using *no* old points at all, just the current solution and derivative. The second order method, which we'll call "AB2," adds the derivative at the previous point into the interpolation mix. We might write the formula this way:

$$y_{k+1} = y_k + h(3f_{\text{ode}}(x_k, y_k) - f_{\text{ode}}(x_{k-1}, y_{k-1}))/2$$

The AB2 method requires derivative values at two previous points, but we only have one when starting out. If we simply used an Euler step, we would pick up a relatively large error on the first step, which would pollute all subsequent results. In order to get a reasonable starting value, we should use the RK2 method, whose per-step error is order $O(h^3)$, the same as the AB2 method.

The following is a complete version of Matlab code for the Adams-Bashforth second-order method.

```
function [ x, y ] = ab2 ( f_ode, xRange, yInitial, numSteps )
% [ x, y ] = ab2 ( f_ode, xRange, yInitial, numSteps ) uses
% Adams-Bashforth second-order method to solve a system
% of first-order ODEs dy/dx=f_ode(x,y).
% f = name of an m-file with signature
%    fValue = f_ode(x,y)
% to compute the right side of the ODE as a column vector
%
% xRange = [x1,x2] where the solution is sought on x1<=x<=x2
% yInitial = column vector of initial values for y at x1
% numSteps = number of equally-sized steps to take from x1 to x2
% x = row vector of values of x
% y = matrix whose k-th row is the approximate solution at x(k).

x(1) = xRange(1);
h = ( xRange(2) - xRange(1) ) / numSteps;
y(:,1) = yInitial;

k = 1;
  fValue =  f_ode( x(k), y(:,k) );
  xhalf = x(k) + 0.5 * h;
  yhalf = y(:,k) + 0.5 * h * fValue;
  fValuehalf = f_ode( xhalf, yhalf );

  x(1,k+1) = x(1,k) + h;
  y(:,k+1) = y(:,k) + h * fValuehalf;

for k = 2 : numSteps
  fValueold=fValue;
  fValue = f_ode( x(k), y(:,k) );
  x(1,k+1) = x(1,k) + h;
  y(:,k+1) = y(:,k) + h * ( 3 * fValue - fValueold ) / 2;
end
```

Exercise 7:

(a) Copy the code to a file called `ab2.m`.

(b) Take a minute to look over this code and see if you can understand what is happening. Insert the following two comment lines into the code in the correct locations:

```
% The Adams-Bashforth algorithm starts here
% The Runge-Kutta algorithm starts here
```

Be sure to include a copy of the commented code in your summary.

(c) The temporary variables `fValue` and `fValueold` have been introduced here but were not needed in the Euler, RK2 or RK3 methods. Explain, in a few sentences, their role in AB2.

(d) If `numSteps` is 100, then *exactly* how many times will we call the derivative function `f_ode`?

(e) Use `ab2` to compute the numerical solution of the ODE for the exponential (`expm_ode`) from `x = 0.0` to `x = 2.0`, starting at `y=1` with step sizes of 0.2, 0.1, 0.05, 0.025, 0.0125 and 0.00625. Recall that the exact solution at `x=2`, is `y=-2*exp(-2)-3`. For each case, record the value of the numerical solution at `x = 2.0`, and the error, and the ratios of the errors. The first line of the table can be used to verify that your code is operating correctly.

			AB2	
numSteps	Stepsize	AB2(x=2)	Error(x=2)	Ratio
10	0.2	-3.28013993	9.4694e-03	__________
20	0.1	__________	__________	__________
40	0.05	__________	__________	__________
80	0.025	__________	__________	__________
160	0.0125	__________	__________	__________
320	0.00625	__________	__________	

(f) Based on the ratios in the table, estimate the order of accuracy of the method, *i.e.,* estimate the exponent p in the error estimate Ch^p. p is an integer in this case.

Adams-Bashforth methods try to squeeze information out of old solution points. For problems where the solution is smooth, these methods can be highly accurate and efficient. Think of efficiency in terms of how many times we evaluate the derivative function. To compute `numSteps` new solution points, we only have to compute roughly `numSteps` derivative values, no matter what the order of the method (remember that the method saves derivative values at old points). By contrast, a third order Runge-Kutta method would take roughly `3*numSteps` derivative values. In comparison with the Runge Kutta method, however, the old solution points are significantly further away from the new solution point, so the data is less reliable and a little "out of date." So Adams-Bashforth methods are often unable to handle a solution curve which changes its behavior over a short interval or has a discontinuity in its derivative. It's important to be aware of this tradeoff between efficiency and reliability!

8 Extra Credit: Stability region plots (12 points)

You have seen above that some choices of step size h result in unstable solutions (blow up) and some don't. It is important to be able to predict what choices of h will result in unstable solutions. One way to accomplish this task is to plot the region of stability in the complex plane. An excellent source of further information about stability regions can be found in Chapter Seven of the book "Finite Difference Methods for Ordinary and Partial Differential Equations" by Randall J. LeVeque at
`http://www.siam.org/books/ot98/sample/OT98Chapter7.pdf`.

When applied to the test ODE $dy/dx = \lambda y$, all of the common ODE methods result in linear recurrance relations of the form

$$y_{k+n} + a_{n-1}y_{k+n-1} + \ldots + a_1 y_{k+1} + a_0 y_k = 0 \tag{7}$$

where $k = 0, 1, \ldots, \infty$, n is some small integer, and the a_j are constants depending on $h\lambda$. For example, the explicit Euler method results in the recurrance relation

$$y_{k+1} - (1 + h\lambda))y_k = 0.$$

It is a well-known fact that linear recursions of the form (7) have unique solutions in the form

$$y_k = \sum_{j=1}^{n} c_j \zeta_j^k$$

where ζ_j for $j = 1, \dots, n$ are the distinct roots of the polynomial equation

$$\zeta^n + a_{n-1}\zeta^{n-1} + \ldots + a_1\zeta^1 + a_0 = 0. \tag{8}$$

If any of the roots is not simple, this expression must be modified slightly. The coefficients c_j are determined by initial conditions. It must be emphasised that the roots ζ are, in general, complex. For the explicit Euler method, for example, $n = 1$ and $a_0 = -(1 + h\lambda)$.

The relationship between (7) and (8) can be seen by assuming that, in the limit of large k, $y_{k+1}/y_k = \zeta$.

Clearly, the sequence $\{y_k\}$ is stable (bounded) if and only if $|y_{k+1}/y_k| \le 1$. Hence the ODE method associated with the polynomial (8) is stable if and only if all the roots satisfy $|\zeta| \le 1$.

The equation (8) can be solved (numerically if necessary) for $\mu = h\lambda$ in terms of ζ. Then, the so-called "stability region," the set $\{\mu : \ |\zeta| \le 1\}$ in the complex plane, can be plotted by plotting the curve of those values of μ with $|\zeta| = 1$. This idea can be turned into the following algorithmic steps.

1. Plug $f_{\text{ode}} = \lambda y$ into the formula you are investigating to yield (8).
2. Write (8) in terms of $\mu = h\lambda$ and $\zeta = y_{k+1}/y_k$
3. For ζ such that $|\zeta| = 1$, solve for μ. This might involve several branches, *e.g.,* when a square root has been taken.
4. For $0 \le \theta \le 2\pi$, set $\zeta = e^{i\theta}$ and draw one or more $\mu(\theta)$ curves. These curves bound the stability region.
5. To identify the *interior* of the stability region, set $\zeta = 0.95e^{i\theta}$ and plot the resulting $\mu(\theta)$ curve(s).

In the following exercise, you will implement this algorithm.

Exercise 8: In the first four parts of this exercise, you will generate the stability region of the explicit Euler method. In the fifth part you will generate the stability region of the Adams-Bashforth AB2 method. The sixth part asks you to interpret the results in the fifth part. The first five parts are worth 8 extra credit points and the sixth part is worth another 4 points.

(a) For real values of θ, the exponential $\zeta = e^{i\theta}$, where $i = \sqrt{-1}$ is the imaginary unit, always satisfies $|\zeta| = 1$. Using 1000 values of θ between 0 and 2π, construct 1000 points on the unit circle in the complex plane. Write a Matlab script m-file to plot these points and confirm they lie on the unit circle. (Recall `axis equal` can be used to get the correct aspect ratio.) Include this plot with the summary of your work.

(b) As remarked above, the explicit Euler method applied to the ODE $dy/dx = \lambda y$ yields $y_{k+1} - (1 + h\lambda))y_k = 0$. Since $(h\lambda)$ occurs together, denote the product as $\mu = h\lambda$. Also denote by ζ the ratio $\zeta = y_{k+1}/y_k$. Writing these expressions using pencil and paper, solve this expression for μ in terms of ζ.

Replace the plot of the unit circle in your Matlab m-file with a plot of the curve $\mu(\zeta)$ for 1000 points on the unit circle $|\zeta| = 1$. Include this plot in your summary file.

(c) Add the curve $\mu(0.95\zeta)$ to your plot to indicate which part of the complex plane represents *stability* instead of instability. Recall you can use `hold on` to plot two curves on the same plot, and you can change the color of the additional curve using the designator `'c'` for cyan or `'r'` for red, *etc.*

(d) Finally, add lines representing the x-axis and y-axis. Send me this plot with your summary.

(e) Make a copy of the Matlab m-file you just wrote and modify it to display *both* the stability region the explicit Euler method *and* the stability region for the Adams-Bashforth AB2 method given above as

$$y_{k+1} = y_k + h(3f_{\text{ode}}(x_k, y_k) - f_{\text{ode}}(x_{k-1}, y_{k-1}))/2$$

Send me this second m-file and the plot with your summary.

(f) (4 additional points) Give examples of odes and values of h that:

   (a) Would be stable using both `ab2` and explicit Euler;

   (b) Would be stable using `ab2` but not for explicit Euler; and,

   (c) Would be stable using explicit Euler but not `ab2`.

   Explain your reasoning and relate it to the plot in the previous part of this exercise.

Comparing the stability regions for explicit Euler and AB2, you can see that oscillatory solutions (whose λ values lie close to the imaginary axis) can be stably simulated using much larger time steps when using AB2 than when using explicit Euler. In contrast, solutions that do not oscillate at all (λ real) can be stably simulated with explicit Euler using larger time steps than with AB2.

Last change `$Date: 2016/12/31 01:40:41 $`