**Laboratorio #10: Métodos Explicitos para Ecuaciones Diferenciales Ordinarias**

**TABLE OF CONTENTS**

* [Introduction](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#INTRODUCTION)
* [The Euler Halfstep Method](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#THE_EULER_HALFSTEP_METHOD)
* [Runge Kutta Methods](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#RUNGE_KUTTA_METHODS)
* [The Midpoint Method](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#THE_MIDPOINT_METHOD)
* [Phase Plane Plots](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#PHASE_PLANE_PLOTS)
* [Adams-Bashforth Methods](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#ADAMS_BASHFORTH_METHODS)
* [ASSIGNMENT](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-10#ASSIGNMENT)

**Introduction**

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.

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 **yk+1**, the approximate solution at the next time step. 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.

**The Euler Halfstep Method**

The Euler halfstep method is a variation of Euler's method. We'll give this method the identifying code **("RK2")**. As part of each step of the Euler halfstep method, an auxiliary solution, which we don't really care about, is computed halfway, using Euler's method:

**xk+1/2 = xk + 0.5 \* dx   
yk+1/2 = yk + 0.5 \* dx \* f(xk,yk)**

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

**xk+1 = xk + dx   
yk+1 = yk + dx \* f(xk+1/2,yk+1/2)**

Surprisingly, although this method uses Euler's method, it ends up having a higher order of convergence. Keep in mind that we do not regard the auxilliary 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.

**SOLVER M FILE**: make a copy of your *euler.m* file, naming it *euler\_halfstep.m*. Modify it to have the form:

**function [ x, y ] = euler\_halfstep ( f, x\_range, y\_initial, nstep )**

Inside the loop, you need to compute two variables called **xhalf** and **yhalf** and use them to define the Euler halfstep solution. If you are not sure what to do here, experiment, and then show me what you have done!

**COMPUTATION**: Compute the numerical solution of the model ODE, (remember? It should be in *model\_ode.m*), from **x = 0.0** to **x = 2.0**, using Euler's method and Euler's halfstep method **("RK2")**, with stepsizes of 0.2, 0.1, 0.05 and 0.025. For each case, record the value of the numerical solution at **x = 2.0**, and the error, that is, the difference between the numerical solution and the true solution at the end point. (A gracious person has given you the results for the first trial, so you can compare with yours.)

Stepsize Euler Euler Error RK2 RK2 Error  
  
 0.2 6.1917 -1.1973 7.3046 -0.0844  
 0.1 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 0.05 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 0.025 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Describe the difference in the way the two sets of errors decrease as the stepsize decreases.

**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 hence of 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 "auxilliary 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 **dx**, computes two intermediate points:

**x1 = x + 0.5 \* dx  
y1 = y + 0.5 \* dx \* f(x,y)  
x2 = x + dx  
y2 = y + dx \* ( 2.0 \* f(x1,y1) - f(x,y) )**

and then estimates the solution as:

**x\_new = x + dx  
y\_new = y + dx \* ( f(x,y) + 4.0 \* f(x1,y1) + f(x2,y2) ) / 6.0**

The global accuracy of this method is **O(h3)**, and so we call it an *order 3* method. Higher order Runge Kutta methods have been computed, with those of order 4 and 5 the most popular.

**COMPUTATION** - In a previous lab, we tried to solve the pendulum problem using the Euler method. We had reason to believe that the solution should stay between -1 and 1, but our computations seemed to blow up. Try to solve the problem again using the order 1 Euler method, and the order 3 RK3 method, over the interval 0 to 25, with initial condition **y**=[1;0].

X Euler RK3  
 0.00 1.00 0.00 1.00 0.00  
 10.00 4.66 4.63 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 20.00 7.43 43.18 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 25.00 -11.69 107.17 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Compare the plots of the two solutions when both use 100 steps.

Moral: Euler's method is not very accurate, even on nice problems. Our solution curve from Euler's method looks smooth, but it's completely incorrect! That's one reason to prefer higher order methods!

**The Midpoint Method**

Your book describes something it calls the "midpoint method", which looks like a slight variation of the Euler halfstep method, but is actually significantly different (and bad). The method needs two sets of initial conditions, at the starting point, and at the first step. We can estimate this second initial condition by using Euler's method on step **k=1**, say. For steps **k=2** and onward, we compute the solution point **k+1** using the derivative at point **k** and the value at point **k-1**:

**xk+1 = xk + dx   
yk+1 = yk-1 + 2 \* dx \* f(xk,yk)**

This really seems very similar to the Euler halfstep method, but it's not the same; now we're keeping the points that were auxilliary before. This may seem a minor change, but there's more to it, and the method can become *unstable*; small errors can quickly grow over successive steps.

As described in the book, the first step taken uses Euler's method. Because the midpoint method is an order 2 method, it would be better if the start-up method was order 2 as well, and the Euler halfstep method could be used.

**FUNCTION M FILE**: make a copy of your problem file *model\_ode.m*, but call it *minus\_ode.m*, and change it so it describes the right hand side of the problem

**y' = - y**

**COMPUTATION** - Copy the midpoint method code *midpoint.m* from the web page. Use it to solve the equation described by the *minus\_ode* M file, on the interval 0 to 2.5, with starting condition **y=1**. You should get the same results as the book:

n Xn Yn Your result  
 0 0.00 1.0000 1.0  
 1 0.25 0.7500 \_\_\_\_\_\_\_\_\_\_\_  
 2 0.50 0.6250 \_\_\_\_\_\_\_\_\_\_\_  
 3 0.75 0.4375 \_\_\_\_\_\_\_\_\_\_\_  
 4 1.00 0.4063 \_\_\_\_\_\_\_\_\_\_\_  
 5 1.25 0.2344 \_\_\_\_\_\_\_\_\_\_\_  
 6 1.50 0.2891 \_\_\_\_\_\_\_\_\_\_\_  
 7 1.75 0.0898 \_\_\_\_\_\_\_\_\_\_\_  
 8 2.00 0.2441 \_\_\_\_\_\_\_\_\_\_\_  
 9 2.25 - 0.0322 \_\_\_\_\_\_\_\_\_\_\_  
 10 2.50 ??? \_\_\_\_\_\_\_\_\_\_\_

Just try to match the data.

**Exercise** - now try the midpoint method, but use a better starting step. In other words, modify the code *midpoint.m*, just in the spot where the single Euler step is taken for **k=1**. Replace this calculation by one step of Euler's halfstep method. Then carry out the same solution procedure as in the previous exercise. You should see the results improve. We're still using the (bad) midpoint method, but we're being more careful to give it a much smaller initial error, and so it behaves better.

**Phase Plane Plots**

In many problems from physics, a system is defined by a second order ODE whose right hand side does not depend on time. Such a system is called *autonomous*. The behavior of the solution doesn't depend on when you start the problem, only on the initial condition. Our pendulum problem is an example of this kind of behavior. Especially for these kind of problems, it can be useful to look at the solution on a special *phase plane plot*. This simply means that instead of plotting **t** or **x** versus the solution **y**, we plot **y** versus **y'**. This is an especially good way to detect periodic solutions, decaying transients, and so on.

**GRAPHICS**: To see the phase plane of an interesting problem, copy the van der Pol ODE code *vanderpol\_ode.m*. This is a famous, simple model of the behavior of the heart. It is a second order system. Copy the *rk3.m* code from the web page, and use it to solve the system. Start with an initial condition of **[ 0; 0.25 ]**, and take maybe 100 steps from 0 to 20. To make a phase plane plot, we want to plot two rows of **y** (or sometimes it's two columns; it depends on how the data is stored by the solver!). For this example, we might try the command

**plot ( y(1,:), y(2,:) )**

Now you should be able to make several guesses about the nature of solutions to this problem! In particular,

* can you guess what will happen if we start with the initial condition **[ 0; 0.00001 ]**?
* What about the initial condition **[ 0; 0]**?
* Will a "large" initial condition like **[0; 5]** settle down?

**Adams-Bashforth Methods**

The Adams-Bashforth methods also 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.

Looked at in this way, the 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:

**xk+1 = xk + dx   
yk+1 = yk + dx \* ( 3 \* f(xk,yk) - f(xk-1,yk-1) ) / 2**

One problem with the AB2 method occurs on the very first step! We need derivative values at two previous points, but we only have one. If we simply used an Euler step, we would pick up a relatively large error on the first step, which would pollute all our results. In order to get a reasonable starting value, we should use the RK2 method, whose per-step error is order **h3**, the same as the AB2 method.

There are two MATLAB issues to think about now. First, can we write our **AB2** routine so that, on the first step, we switch to **RK2**? The other issue is only a question of efficiency. The Adams-Bashforth methods try to save computational time by computing the derivative rarely, and saving the values for later use. Keep these ideas in mind as you look over the following code:

Here is the complete code for **AB2**, the Adams-Bashforth ODE solver. You may copy this code from the web page.

**function [ x, y ] = ab2 ( f, x\_range, y\_initial, nstep )**   
**dx = ( x\_range(2) - x\_range(1) ) / nstep;**   
**x(1) = x\_range(1);**   
**y(:,1) = y\_initial;**   
**yp(:,1) = feval ( f, x(1), y(:,1) );**   
**i = 1;**   
**xhalf = x(i) + 0.5 \* dx;**   
**yhalf = y(:,i) + 0.5 \* dx \* yp(:,1);**   
**yphalf = feval ( f, xhalf, yhalf );**   
**x(i+1) = x(i) + dx;**   
**y(:,i+1) = y(:,i) + dx \* yphalf;**   
**yp(:,i+1) = feval ( f, x(i+1), y(:,i+1) );**   
**for i = 2 : nstep x(i+1) = x(i) + dx;**   
**y(:,i+1) = y(:,i) + dx \* ( 3.0 \* yp(:,i) - 2.0 \* yp(:,i-1) ) / 2.0;**   
**yp(:,i+1) = feval ( f, x(i+1), y(:,i+1) );**   
**end**

**MEDITATION**: Take a minute to look over this code and see if you can understand what is happening. Identify the Runge-Kutta step. What new important data is being saved in this code and why? If **NSTEP** is 100, then *exactly* how many times will we call the derivative function **f**?

**COMPUTATION**: Copy the code *ab2.m* from the web page. Use it to compute the numerical solution of the model ODE from **x = 0.0** to **x = 2.0**, with stepsizes of 0.2, 0.1, 0.05 and 0.025. For each case, record the value of the numerical solution at **x = 2.0**, and the error.

Stepsize AB2 AB2 Error  
  
 0.2 7.1927 0.1964  
 0.1 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 0.05 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 0.025 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

For this example, what is the behavior of the error with decreasing stepsize? When we concentrate on the error at a fixed position, are we looking at the *global* or the *local* error of the method?

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 **N** new solution points, we only have to compute **N** derivative values. By contrast, a third order Runge-Kutta method would take **3\*N** 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!

**Assignment**

The third order Adams-Bashforth method "AB3" can be summarized in the formula:

**Yk+1 = Yk + dx \* ( 23 Y'k - 16 Y'k-1 + 5 Y'k-2 ) / 12**

Make an M file called *ab3.m*, and figure out how to implement the AB3 method. Use an RK3 method to get enough starting values. Let me take a look at your code to make sure you have the right idea!

**Assignment**: Use your AB3 code to compute the numerical solution of the model ODE, from **x = 0.0** to **x = 2.0**. Record the value of the numerical solution at **x = 2.0**, and the error (the difference between **y(2.0)** and exp(2.0)).

dx y(2.0) y(2.0) - exp(2.0)  
  
 0.2 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_   
 0.1 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_   
 0.05 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_   
 0.025 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Send mail to me containing

* the results of this computation (eight numbers);
* your guess for the rate of convergence of this method. (the error at **x=2.0** behaves like a constant, or **dx**, or **dx2** or **dx3**...?)