**Laboratorio # 9: Métodos Implicitos para Ecuaciones Diferenciales Ordinarias**

**TABLE OF CONTENTS**

* [Introduction](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#INTRODUCTION)
* [Stiff Systems](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#STIFF_SYSTEMS)
* [The Backward Euler Method](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#THE_BACKWARD_EULER_METHOD)
* [The Trapezoid Method](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#THE_TRAPEZOID_METHOD)
* [The Starting Point Problem](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#THE_STARTING_POINT_PROBLEM)
* [BDF Methods](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#BDF_METHODS)
* [Direction Field Plots](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#DIRECTION_FIELD_PLOTS)
* [Summary](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#SUMMARY)
* [ASSIGNMENT](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-9-1#ASSIGNMENT)

**Introduction**

The explicit methods that we discussed last time are well suited to handling a large set of ODE's. The Adams-Bashforth methods are very inexpensive, using old data to predict the next value. To get a higher order method, we simply use more data. The Runge-Kutta methods have a complicated form, and are expensive, because they evaluate the derivative function at lots of extra points, but these methods can be a bit more accurate, since they are sampling the derivative function right near the place where the next point is to be computed. And high-order versions of Runge-Kutta methods are available to give very accurate results.

This two sets of methods perform poorly, however, for a class of "stiff" problems, that can come up in chemistry and biology. We will examine two families of *implicit methods* suitable for such problems. We will find that the implementation of an implicit method has at least two extra complications we didn't see with the explicit method: getting a starting value, and solving (or not solving!) the nonlinear equation.

**Stiff Systems**

I've warned you that there are problems which defeat the explicit Runge-Kutta and Adams-Bashforth methods. In fact, for such problems, the higher order methods perform even more poorly than the low order methods. These problems are called "stiff" ODE's. We will only look at some very simple examples.

Consider the differential equation

**y' = - 5 \* y  
y(0) = 1**

whose solution is **y(x) = exp ( - 5 \* x )**. This function is continuous, infinitely differentiable, and monotone. So let's try to solve it!

**FUNCTION M FILE**: Make a copy of *model\_ode.m*, called *stiff\_ode.m*, which has the form

**function yprime = stiff\_ode ( x, y )  
yprime = - 5 \* y;**

**COMPUTATION**: Compute the numerical solution of the stiff ODE, from **x = 0.0** to **x = 2.0**. Record the value of the solution at **x = 2.0**. Try the Euler method with several stepsizes. Try several methods at the same stepsize. Plot your results and compare them visually to the exact solution.

Stepsize Euler AB2 RK3   
  
 0.2 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_   
 0.1 \_\_\_\_\_\_\_\_\_\_   
 0.05 \_\_\_\_\_\_\_\_\_\_   
 0.025 \_\_\_\_\_\_\_\_\_\_

For the stepsize 0.2, which method is the "least bad", in terms of the error at the end?

**The Backward Euler Method**

The Backward Euler method is a variation of Euler's method. Before we say anything more about it, let's take a long hard look at the algorithm:

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

If you sneeze at the wrong time, you might think there is no difference between this method and Euler's method. But look carefully. This is *not* a "recipe", the way most formulas are. The equation defining **yk+1** is implicit, and has to be handled in a more careful way than we are used to. So our instinctive reaction might be, well who needs this kind of complication? It happens that there are ODE's for which this approach is the best known way to produce a solution, where Runge-Kutta and Adams-Bashforth methods would produce only gibberish.

Another thing to notice about this method is that it's exactly Euler's method "time-reversed". In other words, if we started at **(x0,y0)**, with a given differential equation, then we could use Euler's method to go "right" to **(xn,yn)**, or we could start at the left endpoint and use the backward Euler method to go back to the starting point.

But to use the backward Euler method in a real situation, we have to deal with the implicit nature of the formula. In fact, what most algorithms do is "fudge" it. They use a reasonably good guess for what the answer is, make one small adjustment that they hope is an improvement, and declare victory. This process is so common that it has a name, and these methods are often called *predictor-corrector* methods.

So now let's look at a very simple implementation of the backward Euler method. We are going to treat the formula as a machine. We dump an estimated value of **yk+1** into the right hand side of the formula, and out comes a "corrected" estimate. Where do we get our initial estimate? We use the forward Euler method, as a "predictor". Now generally this is not going to produce a quantity that actually satisfies the backward Euler formula. If we put the corrected value in the right hand side, we'd get yet another value out, and so on. The hope is, that at least for small enough steps, this is a convergent process and that one corrector step might be "good enough".

This may sound like a pretty rickety idea, but it's a start. This technique of handling an implicit ODE approximation formula by a prediction step and a single correction step is called *PECE*. (The *E's* represent evaluations of the derivative function).

Here's what our PECE version of the backward Euler method looks like:

**xk+1 = xk + dx   
yp = yk + dx \* f(xk,yk) *(predict)*  
yk+1 = yk + dx \* f(xk+1,yp) *(correct)***

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

**function [ x, y ] = be\_pece ( f, x\_range, y\_initial, nstep )**

Now change the code to carry out the backward Euler method, using the prediction/correction approximation described above. The easiest way is to define and use a variable **yp** in the obvious way.

**COMPUTATION**: Compute the numerical solution of the model ODE, from **x = 0.0** to **x = 2.0**, using the backward Euler PECE method, 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**. (The exact value is exp(2.0) = 7.389056, and the Euler method values are included for comparison.)

Stepsize Euler BE\_PECE   
  
 0.2 6.191736 \_\_\_\_\_\_\_\_\_\_   
 0.1 6.727500 \_\_\_\_\_\_\_\_\_\_   
 0.05 7.039989 \_\_\_\_\_\_\_\_\_\_   
 0.025 7.209566 \_\_\_\_\_\_\_\_\_\_

Can you judge the rate of convergence of this method?

**COMPUTATION**: Repeat the previous computation, but now do it for *stiff\_ode*, whose exact solution at 2 is exp(-10.0) or about 0.000045:

Stepsize BE\_PECE   
  
 0.2 \_\_\_\_\_\_\_\_\_\_   
 0.1 \_\_\_\_\_\_\_\_\_\_   
 0.05 \_\_\_\_\_\_\_\_\_\_   
 0.025 \_\_\_\_\_\_\_\_\_\_

We said that the backward Euler method was stable, which means roughly that we don't expect errors to grow, or for our solution to blow up "unnecessarily". But your results probably look very bad. Can you explain how the backward Euler method can be good and your results bad? (Hint: we are not exactly using the backward Euler method! That is, we are not using the backward Euler method...exactly.)

**COMPUTATION**: From the web page, copy the files: *stiff\_ode\_partial.m* and *be\_newton.m*. In order to solve the problem with this method, you need to use a slightly more complicated command, such as:

**[x,y] = be\_newton ( 'stiff\_ode', 'stiff\_ode\_partial', [0,2], 1, 10 );**

Repeat the previous computation using this new version of the backward Euler method:

Stepsize BE\_NEWTON   
  
 0.2 \_\_\_\_\_\_\_\_\_\_   
 0.1 \_\_\_\_\_\_\_\_\_\_   
 0.05 \_\_\_\_\_\_\_\_\_\_   
 0.025 \_\_\_\_\_\_\_\_\_\_

These results should seem more reasonable. Can you guess what information the extra routine *stiff\_ode\_partial.m* supplies, and how that information is used?

**The Trapezoid Method**

The trapezoid method can be derived from the trapezoid rule for integration. It has a simple form:

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

from which you can see that this is also an *implicit* formula. The backward Euler and Trapezoid methods are the first two members of the *Adams-Moulton* family of ODE solvers.

To properly use the trapezoidal formula, we must be prepared to solve a nonlinear equation. But it is very common to use a watered-down version of the formula, which is not guaranteed to have the good stability properties, but which is easy to solve. Instead of using Newton's method, we can use the method of *successive substitution* or *functional iteration*. In such a method, we have a nonlinear equation which can be written as

**y = f(y)**

and we "solve" it by finding an estimate for **y**, and then repeatedly using the formula, that is, replacing **y** by the value of **f(y)**. We won't even try to make the process converge; we'll just take two steps and be hopeful. And we'll plan to use the forward Euler method to get our approximate starting point.

With these modificiations, our version of the trapezoidal method can be termed a "PECECE" method, and can be written:

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

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

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

To turn it into the PECECE version of the trapezoid method, you simply need to follow the prediction line that is already there by two corrections. (The fact that PECE methods are so easy to implement makes them popular, even though they are not "true" implementations of the implicit methods.)

**FUNCTION M FILE**: make a copy of your *model\_ode.m* file, naming it *bump\_ode.m*. It should evaluate the right hand side of the ODE

**y' = - y2**

When the initial condition is Y(0) = 1, then the exact solution is **y(x)=1/(1+x)**

**COMPUTATION**: Test your method by computing the numerical solution of *bump\_ode.m*, using initial condition **y(0)=1**, to **x = 5.0**, using the trapezoid PECECE method, with stepsize 0.25. Record the value of the numerical solution at regularly spaced intervals. The book carried out this experiment, and we list the values it gives.

X Exact Y Book Y TRAPEZOID\_PECECE  
  
 0.0 1.000 1.000 1.0  
 1.0 0.500 0.496 \_\_\_\_\_\_\_\_\_\_   
 2.0 0.333 0.331 \_\_\_\_\_\_\_\_\_\_   
 3.0 0.250 0.249 \_\_\_\_\_\_\_\_\_\_   
 4.0 0.200 0.199 \_\_\_\_\_\_\_\_\_\_   
 5.0 0.167 0.166 \_\_\_\_\_\_\_\_\_\_

If your code is correct, you should get very close to the book's approximate values!

**The Starting Point Problem**

If we actually try to solve the nonlinear equation defining an implicit formula, then it doesn't matter too much what initial value we use, as long as we're close enough for the nonlinear equation solver to converge. So I could use an Euler prediction with *be\_newton*, or even just the old value of the solution, **yk**.

However, in a PECE method (or PECECE, or whatever) method, where we only take a few corrector steps, a poor starting value may mean the corrector can't get us a good result. So it becomes more important to try to get a reasonably good predictor method. What is common practice, when using an Adams-Moulton corrector, is to use the Adams-Bashforth method of the same order as the predictor. Thus, we might be better off in the trapezoid rule using the AB2 method for the predictor...but that's not what the book does. This kind of predictor/corrector method is called an *Adams-Bashforth-Moulton* method or **ABM**.

**COMPUTATION**: Try out a simple PECE version of an Adams-Bashforth-Moulton method of order 3. Copy the code *abm3\_pece.m* from the web page. Test the method on the *bump\_ode.m* problem, using initial condition **y(0)=1**, to **x = 5.0**, with stepsize 0.25.

X Exact Y ABM3\_\_PECE  
  
 0.0 1.000 1.0  
 1.0 0.500 \_\_\_\_\_\_\_\_\_\_   
 2.0 0.333 \_\_\_\_\_\_\_\_\_\_   
 3.0 0.250 \_\_\_\_\_\_\_\_\_\_   
 4.0 0.200 \_\_\_\_\_\_\_\_\_\_  
 5.0 0.167 \_\_\_\_\_\_\_\_\_\_

Take a look at the code itself, and examine how it handles the starting point problem.

**BDF Methods**

BDF methods (backward differentiation formulas) are similar to Adams-Moulton methods, in that they are implicit, and intended to be stable in the face of stiff problems. While the backward Euler method qualifies as the first member of both of these families, the two families differ in that the Adams-Moulton methods look back at values of the derivative **y'** at old points, while the BDF methods look back at values of **y** itself. *(This makes a BDF method easier to use with a variable step size, which we haven't even started to think about yet!)*

The second member of the BDF family, "BDF2", can be summarized as

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

Because of the way I have written this formula, you might recognize that it is using 1/3 of a finite difference estimate of the derivative between points **k-1** and **k**, and 2/3 of the derivative at point **k+1**. But in particular, the only place where **f** is evaluated is at the new point. Compare this to the trapezoid method, which uses several old derivative values, but only one old solution value. The same distinction continues between **BDF3** and **AM3** and so on.

BDF2 is implicit, and so there are the same problems of getting a starting point, and deciding how well to solve the nonlinear equation at each step, as with the backward Euler method. A new problem occurs, however, because we can't use BDF2 until we have two old solution values. *(This is the same problem we had with the explicit multistep methods, like* ***AB2****)*.

When we had this startup problem with the Adams-Bashforth methods, we simply used a Runge-Kutta method. But with a stiff problem, even one **RK2** step may get us into trouble! So for now, let's decide only to use implicit methods. Then we're stuck with using a first step of **BDF1** before we can switch to our desired method of **BDF2**. *(If stability is really a problem, and we want to use a BDF method of order* ***N****, but we don't want to lose accuracy, our first* ***N-1*** *steps could be very small* ***BDF1*** *steps, assuming that our* ***BDF-N*** *method can handle a variable stepsize)*.

**COMPUTATION**: copy *bdf2\_newton.m* from the web page. Try your code out on the stiff ODE, going from **x** = 0 to 2, with initial condition **y** = 1. Compare your results with the results from *be\_newton.m*. You should expect better accuracy.

**Direction Field Plots**

One way of looking at a differential equation is to plot its *direction field*. At any point **(x,y)**, we can plot a little arrow in the direction that the differential equation is telling us to go. This is something like the wind direction. What is that direction? Think of how the Euler method works: if you take a step of **dx** in the **x** direction, then you must move **(dy/dx)\*dx** in the **y** direction. In other words, the direction vector is some multiple of **(dx,(dy/dx)\*dx)**.

So to make a direction field plot of, say, our stiff ODE, we might want to pick the points **(x,y)** in the unit square that are spaced **dx** = 1/10 apart. The MATLAB *meshgrid* command will help us do that. Then we need to evaluate the direction vector **(u,v)**. The **u** vector will be all 1's (use the MATLAB *ones* function), and the **v** vector comes from our right hand side function. We probably want to use a scale factor of about **dx** to keep the arrows from getting muddled. Finally, we use the special MATLAB command *quiver* to display the vector plot. *(I don't think "quiver" is available in OCTAVE!)*. Here's how to plot the direction field for the [0,2] by [-1,1] range of **(x,y)** values:

**dx = 0.1;  
[xp,yp] = meshgrid ( 0:dx:2, -1:dx:1 );  
u = dx \* ones ( size ( xp ) );  
v = dx \* stiff\_ode ( xp, yp );  
quiver ( xp, yp, u, v )**

*(I'm calling the points "[xp,yp]" to distinguish them from the "[x,y}" that we use to store our ODE solution.)* If the arrows aren't the right size, you can make them bigger (or smaller):

**u = 2 \* u;  
v = 2 \* v;  
quiver ( xp, yp, u, v )**

You may also want to issue the command

**axis equal**

to force the plot to be in the correct proportions. Otherwise, you can't be sure a diagonal line is really diagonal!

**GRAPHICS**: Use the above commands to make a plot of the direction field of the stiff ODE. Compute the solution of the ODE from 0 to 2, using 10 steps, and the *bdf2\_newton* code. Issue the *hold on* command, and then plot your solution on top of the direction field. Can you see why this is a hard problem to solve?

*If the differential equation is higher order, then it's more difficult to figure out a sensible way of plotting a direction field. Consider, for instance, how you would try to plot the direction field for the predator-prey problem! However, that's a nice problem for phase plane plots!*

**Summary**

It can be hard to keep track of the different ODE methods and their properties. Here is a summary of the four important families of numerical ODE solvers we have discussed in the labs and in the class.

* **Runge-Kutta** methods are explicit, and require **yk** and **y'k**, and values of **y** and **y'** at auxilliary points between **yk** and **yk+1**. (RK1 = Euler's method, RK2 = Euler Halfstep, RK3, ...)
* **Adams-Bashforth** methods are explicit, and require **yk**, and values of **y'** at previously computed points **k**, **k-1**, ... (AB1 = Euler's method, AB2, ...)
* **Adams-Moulton** methods are implicit, and require **yk**, an estimate of **yk+1**, and values of **y'** at points **k+1**, **k**, ... (AM1 = Backward Euler Method, AM2 = trapezoid method, AM3, ...)
* **BDF** methods are implicit, and require an estimate of **yk+1**, the value of **y'k+1**, and values of **y** at points **k**, **k-1**, .... (BDF1 = Backward Euler Method, BDF2, ...)

The implicit methods require a starting estimate of the new solution, and also are generally nonlinear equations. When using a particular implicit method, there are many ways to make the prediction, and many ways to decide how to handle the nonlinear equation (PECE, functional iteration, Newton's method).

The midpoint method doesn't really fit into any of these categories. It's an explicit method, but it's not an Adams-Bashforth method, because it uses an older solution value, **yk-1**.

**Assignment**

Compute the numerical solution of the model ODE, from **x = 0.0** to **x = 2.0**, using the trapezoid PECECE method, with the given stepsizes. Record the value of the numerical solution at **x = 2.0**, and 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. (As **dx** descreases, does the error at **x=2.0** behave like a constant, or **dx**, or **dx2** or **dx3**...?)