**Laboratorio #11**

**TABLE OF CONTENTS**

* [Introduction](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#INTRODUCTION)
* [Boundary Value Problems](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#BOUNDARY_VALUE_PROBLEMS)
* [Shooting Methods](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#SHOOTING_METHODS)
* [Discretizing a BVP](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#DISCRETIZING_A_BVP)
* [A Heat Equation](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#A_HEAT_EQUATION)
* [The Method of Lines](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#THE_METHOD_OF_LINES)
* [A Wave Equation](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#A_WAVE_EQUATION)
* [ASSIGNMENT](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-11#ASSIGNMENT)

**Introduction**

Our study of the initial value problem for ordinary differential equations has taught us how to estimate the behavior, over time, of a scalar quantity, supposing we have an initial value of the quantity, and a description of how it tends to change over time. This very limited ability can nonetheless sometimes be extended to a few other kinds of problems.

One class of problems is known as *boundary value problems*. A simple example of such a problem would describe the shape of a chain hanging between two posts. We know the position of the endpoints, and we have a second order differential equation describing the shape. If the two conditions were both given at the left endpoint, we'd know what to do right away. But how do we handle this "slight" variation?

The other sort of problem that we can take a stab at is the solution of certain *partial differential equations*, particularly equations that describe the flow of heat. We will also show that certain kinds of wave equations can be handled in this way. Really interesting problems require more sophisticated treatment, but at least we can get some flavor of a new field of applications!

**Boundary Value Problems**

A one-dimensional boundary value problem, or "**BVP**", is similar to an initial value problem, except that the data we are given isn't conveniently located at a starting time, but rather some is specified at the left end point and some at the right. (We're also usually thinking of the independent variable as representing space, rather than time, in this setting).

Our ideas about solving an initial value problem, or "**IVP**", have to be modified in some way. One possibility is to stubbornly use the **IVP** idea, and make up any necessary initial conditions. The other idea might be to somehow think of the solution points of the discretized equation as satisfying a set of coupled linear equations that we can solve.

**The clothesline BVP**: a rope is stretched between two points. If the rope were weightless, or rigid, it would lie along a straight line; however the rope has a weight and is elastic, so it sags down slightly from its ideal linear shape. Determine the curve described by the rope.

If we define **y(x)** to be the vertical position of the rope at **x**, then a simple model for this problem is:

**y'' = 0.1  
y(0.0) = 0  
y(5.0) = 0.5**

The number **0.1** is a physical constant that describes how easy it is for the rope to stretch in response to gravity. Different values describe different ropes. Ignoring the boundary conditions, this ODE is *linear*. This actually implies a lot of things about the existence and uniqueness of solutions.

**M FILE**: write an M file called *rope\_ode.m* which evaluates the right hand side vector for the first order linear BVP system. (We're not worried about the boundary conditions just yet).

**The chemical reaction BVP**: the concentration of a certain reactant is supposed to satisfy the following equation:

**u'' = lambda \* eu  
u(0.0) = 0.0  
u(1.0) = 0.0**

Here **lambda** is a physical constant to be specified later. This ODE is not linear. A solution of the original ODE is not guaranteed to exist; if a solution exists, it is not guaranteed to be unique. These facts affect the numerical solution of the problem.

**M FILE**: write an M file called *chemical\_ode.m* which evaluates the right hand side vector for the first order chemical BVP system. Include a line setting **lambda** to -1, but be prepared to change this value!

**Shooting Methods**

Since we don't have any idea how to solve a BVP, but we know how to solve an IVP, let's forget the rules and just do what we do best. Taking as our example the rope BVP, how much violence do we have to do in order to make it look like an IVP? Well, we expect to have *two* conditions at the left point and none at the right point. So let's temporarily consider the related problem, where we have made up an extra boundary condition at our "initial" value of **x**:

**y'' = 0.1  
y(0.0) = 0.0  
y'(0.0) = ALPHA**

**COMPUTATION**: Some value of **ALPHA** will *probably* give us a solution which has the desired value **y(5.0)=0.5**. Try a few values of **ALPHA**. I suggest using **NSTEP**=100, with the **RK3** method. The first line lists one result I got:

Given Left Guess Right Error  
  
 Y(0.0) Y'(0.0) = ALPHA Y(5.0) - 0.5  
  
 0.0 2.0 11.25 - 0.5 = 10.75  
 0.0 ------------- -------------------  
 0.0 ------------- -------------------

This is very much like the "cannon ball" problem you had a few labs ago! The difference is that now we want to figure out a *systematic* way of having the computer do this problem!

Convince yourself that:

* For each value of **ALPHA**, we can solve this problem, and get a numerical solution at a sequence of points up to the right endpoint.
* Since every value of **ALPHA** determines a (numerical) solution **y(5.0)**, we can regard the difference between the value we got, and the value we want, as a function **f(ALPHA) = y(5.0) - 0.5**.
* The BVP solution we are looking for has the property that **f(ALPHA)** = 0.
* We can use a nonlinear equation solving algorithm to seek a suitable value of **ALPHA**.

Now the easiest way to set up a solution scheme for this problem would be to get two solutions to start with, and then use the secant method to try to locate a good solution. So our algorithm might be:

1. For **ALPHA** = 0.0, compute **y(5.0)**, and then **f(0.0) = y(5.0) - 0.5**;
2. For **ALPHA** = 1.0, compute **y(5.0)**, and then **f(1.0) = y(5.0) - 0.5**;
3. Use the secant method and the previous two values of **ALPHA** and **f(ALPHA)** to compute the next value of **ALPHA** to try.
4. Evaluate **f(ALPHA)**. If small enough, return the value of **ALPHA**.
5. If we've taken more than 20 steps, print an error message and give up.
6. Go back to step 3.

Where we write **y(5.0)**, we should really write **y(5.0,ALPHA)**, to emphasize that the solution depends on the parameter.

*Things you should think about*: What is it about the bisection method that made me not choose that as my nonlinear equation solver? What disadvantage can you see to using Newton's method? Can you think of at least two ways in which the secant method could break down, even if there really was a solution to the problem?

**M FILE**: write an M file called *rope\_shoot.m* which accepts values of **ALPHA** and **NSTEP** and evaluates **f(ALPHA)**, for the rope BVP. Your code should have the form:

**function f = rope\_shoot ( alpha )**

and this code should do the following:

* Use the value of **alpha** to define the second initial condition;
* Call *rk3.m* to compute the solution **[x,y]** of the IVP defined by the initial conditions, and the right hand side function *rope\_ode*, using **NSTEP**=100 steps;
* Return in the function value **f** the value of **y(1,NSTEP+1)**.

**COMPUTATION**: Copy the code *secant.m* from the web page. (This version of the secant method has been rewritten from what we used last term.) Use this secant method, and the ideas discussed above, to seek a solution of the rope BVP. What value of ALPHA did your algorithm discover? Plot your solution to see if the rope looks like it is hanging properly, and has the correct position at the right endpoint.

**Discretizing a BVP**

For the rope problem, we know that the ODE determines the position of the rope. Rather than think of this as happening by means of a formula, it's better to think of the ODE as reporting, at each point **x**, that the "curvature" **y''** of the rope balances the force **0.1**. (The second derivative is similar to curvature, but it is a mathematical sin to pretend that it is the same thing).

Keep this idea in mind. Assume that we have divided the interval up into **N-1** equal intervals of width **dx** determined by **N** points, and now use the standard central difference approximation to the second derivative. The difference equation that corresponds to our ODE is then:

**( yi-1 - 2 yi + yi+1 ) / dx2 = 0.1**

We can associate this equation with the solution value at **yi**, except for **i=1** and **i=N**. But magically, those just happen to be the points at which we have boundary conditions specified.

In particular, let us look at approximating our rope BVP at 6 points. We set up the ODE at points 2, 3, 4 and 5, and associate the boundary conditions with the first and last solution values. I also multiplied through by the divisor to make things look nicer:

u1 = 0.0  
 u1 - 2 u2 + u3 = 0.1 \* dx2  
 u2 - 2 u3 + u4 = 0.1 \* dx2  
  
 u3 - 2 u4 + u5 = 0.1 \* dx2  
 u4 - 2 u5 + u6 = 0.1 \* dx2  
  
 u6 = 0.5

As a sanity check, what are these equations saying? The boundary conditions "nail down" the ends of the rope. But what is this relationship between sets of three successive points? Well, ignore the right hand side, and ask yourself, what is the typical value of **yi-1-2yi+yi+1** for points on a constant function, a linear function, a quadratic function? You should see that this quantity (divided by **dx2**) will exactly compute the second derivative of functions up to a quadratic. And conversely, if you know the second derivative, you expect a particular relationship between consecutive triples.

By discretizing the differential equations we have created a set of linear alebraic equations, which have the symbolic form **A\*u=b**. (Officially, we don't have any idea what linear equations are, because that's in the *next* chapter of the book. But unofficially, let's keep going.) To set up and solve these equations in MATLAB, we could type:

**n = 6;**  
**x = linspace ( 0.0, 5.0, n );**  
**A = [ 1 0 0 0 0 0; 1 -2 1 0 0 0; 0 1 -2 1 0 0; 0 0 1 -2 1 0; 0 0 0 1 -2 1; 0 0 0 0 0 1];**  
**dx = 5.0 / ( n - 1 );**   
**rhs = 0.1 \* dx^2;**   
**b = [ 0.0; rhs; rhs; rhs; rhs; 0.5 ]; *(a column vector!)***  
**u = A \ b**

You probably recognize everything that's happening here except for the last bizarre line. That line essentially tells MATLAB to solve the equation **A\*u=b**. We'll have to wait til the next chapter to really talk about that!

**Computation**: Try to solve the rope BVP using this method. Print out the vector **b** and verify by hand at least one of the linear equations. Then plot **x** versus **u**.

**Computation**: To get more accuracy, we want to increase **N**, the number of points. Consider the problem of typing in the matrix **A** if **N** is 101! Since the matrix is so simple, we can use the MATLAB **diag** command to set up the "1 -2 1" diagonals, although we will have to adjust the first and last row. Copy the file *dif2.m* from the web page. Type the command **A = dif2 ( 6 )**. Now correct the first and last rows of **A**, namely, the entries **A(1,1), A(1,2), A(N,N-1)** and **A(N,N)**, to get the matrix we had in the previous problem.

*Got a minute for a puzzle?* The function **ones(m,n)** returns an **m** by **n** matrix of 1's. The function **diag(v,d)** returns a matrix whose **d**-th diagonal contains the values in the vector **v**. Using these ideas, can you figure out how **dif2.m** can compute the difference matrix in a *single* line? OK, you can look now!

**Computation**: Now we are ready to solve the big problem! Repeat the previous calculation, but now with **N=101**. Use the **dif2** function to set up the matrix **A**, and correct the first and last rows. You must also figure out a clever way of entering the 101 entries of **b**. Can you see how the **ones** function can help you again? Now solve the linear system, and plot **x** versus **u**.

If you call your problem variables **u2** and **x2**, you can compare the plots of **(x,u)** and **(x2,u2)** and see how refining the mesh has changed the solution.

**A Heat Equation**

A partial differential equation or "PDE" involves derivatives of a function **U** which depends on more than one independent variable. One of the classic PDE's is the one-dimensional *heat equation*, which describes the behavior of the temperature **U(x,t)** of a rod.

**Ut = Uxx + f(x,t)**

We also specify an initial condition, that is, the temperature of every point on the rod at the starting time,

**U(x,0) = g(x)**

and boundary conditions, describing the behavior of the solution at points which are exposed to the outside:

**U(0,t) = 0,  
U(1,t) = 0**

The boundary conditions could be more interesting, changing with time, or describing the behavior of the derivative **Ux**.

**The Method of Lines**

To find a numerical approximation to the solution of the heat equation, one approach is to focus attention on a finite grid of equally spaced points, **(x1,...,xN)**. Having discretized space in this way, we know that at any fixed time **t**, the solution function will now be a vector of values **(U(x1,t),...,U(xN,t))** associated with these grid points. We must replace our differential equation by an appropriate equation involving spatial differences of these solution values, which we still think of as having a continuous dependence on time.

Without yet discretizing time, our ODE has become the following semi-discretized system for **(U(x1,t),...,U(xN,t))**.:

**(d/dt) U(xi,t) = (U(xi-1,t)-2\*U(xi,t)+U(xi+1,t))/dx2 + f(xi,t)**

These equations may be set up for **i** = 2 through **N-1**. Our first and last equations will use the boundary conditions, but can you see how the first equation, **U(x1,t)=0**, can be made to look like a differential equation? The right hand boundary condition can be treated this way as well. So now we have **N** ordinary differential equations.

**Example**: Consider the PDE:

**Ut = Uxx + x\*t   
U(x,0) = sin(pi\*x)   
U(0,t) = 0, U(1,t) = 0**

If we discretize space using just 4 points, and replace the differential operator by a discretized version, *and* simplify notation by writing **U(i,t)** when we really mean **U(xi,t)**, we get:

**d/dt U(1,t) = 0   
d/dt U(2,t) = (U(1,t)-2\*U(2,t)+U(3,t)) / (1/9) + 0.33 \* t   
d/dt U(3,t) = (U(2,t)-2\*U(3,t)+U(4,t)) / (1/9) + 0.67 \* t   
d/dt U(4,t) = 0**

with initial conditions:

**U(1,0.0) = 0.0   
U(2,0.0) = 0.5 \* sqrt ( 3.0 )   
U(3,0.0) = 0.5 \* sqrt ( 3.0 )   
U(4,0.0) = 0.0**

and we know how to solve a system of ODE's like this! This method of discretizing in space to turn a PDE into a set of coupled ODE's is called the *method of lines*. It is only applicable to certain classes of PDE's. The stepsize **dt** used in the time discretization often must satisfy a certain stability test defined by the spatial discretization size **dx**. In short, once you've picked your spatial discretization, there is a limit on the size of your time step.

If we are using Euler's method for the time integration, and if our spatial stepsize is **dx**, then stability suggests using a time step of **0.5\*dx2**. If we do so, then we may expect that each time we halve **dx**, the error will go down by a factor of 4. We take the error at the midpoint as representative of the total error.

Consider the following PDE:

**Ut = Uxx + (pi2-0.1) \* exp (-t/10) \* sin(pi\*x)   
U(x,0) = sin(pi\*x)   
U(0,t) = 0, U(1,t) = 0**

which has the exact solution:

**U(x,t) = exp(-t/10) \* sin(pi\*x)**

**Computation**: Copy the file *method\_of\_lines.m* from the web page. Type **help method\_of\_lines** to see how to use it. Use the method of lines with various values of **nx**, and **nt**, to compute a table of values of the PDE solution **U** for **0.0 <= x <= 1.0** and **0.0 <= t <=5.0**.

NX NT Computed Exact  
 (space) (time) U(0.5,5.0) U(0.5,5.0) Error  
 4 160 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 8 640 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 16 1280 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

(If your PC can't handle it, you can skip the last computation. But if you can solve it, what goes wrong? How can we fix it?) When we write the functional form **U(0.5,5.0)**, where do we actually need to look in the MATLAB variable **U(?,?)**.

To see a snapshot of the solution at a timestep **j**, use the command

**plot ( x, u(:,j) )**

To see the time evolution of the solution value at a fixed point **i**, use the command

**plot ( t, u(i,:) )**

If you're using MATLAB, one way to get a nice plot of the entire solution is to type

**mesh ( u )**

**A Wave Equation**

PDE's describing the behavior of waves are superficially similar to those describing heat flow, but the properties of the equation and its solution are usually very different. We do not have time to worry about these matters; we will simply run an example in which a version of the method of lines is used successfully to solve a wave equation.

Suppose we are observing the behavior of a one-dimensional ocean. Waves may arise, or move to the left or right. We keep track of the quantity **u(x)**, the height of the water surface, at each point. In the absence of any other disturbances, we know that waves will move about in more complicated way than heat does. One model for this behavior is the *Korteweg-deVries equation* or *KdV* equation:

**du/dt = 6 \* u \* du/dx - d3u/dx3.**

As with the heat equation, we are going to discretize space, then discretize the spatial differentiation operators, and come up with a semi-discrete coupled system of differential equations. There are a couple of differences you should note. The second derivative is easy to approximate with a second difference that is *symmetric*. But the commonly used approximations to first and third derivatives are not symmetric. In order to avoid adding a directional bias in the equation, we will go out of our way to use symmetric approximations. Secondly, instead of using the Euler method to do the time integration, we will use a version of the midpoint method.

If you are running MATLAB, copy the M files *kdv\_mid.m* and *kdv\_movie.m*. Otherwise, copy just the first file. Look at how the calculation in *kdv\_mid* is set up. Be able to answer the following:

* How can we tell this is a method-of-lines code?
* What formulas are used to approximate the first and third space derivatives?
* Why is there a factor of 2 multiplying most of the right hand side in the midpoint rule computation of **u(it,ix)**?
* At each new time, the first two and last two values of **U** are not approximated by an ODE, but are instead copied from the previous time step. Why is this done, and why must we do this for the first and last *two* values? (In the heat equation, we only handled one value at each endpoint this way.)

If you are running MATLAB, run *kdv\_movie*, otherwise run *kdv\_mid*. For a change, don't bother thinking. Just enjoy looking at the movie.

If you were able to run the movie, then you will see a simulation of the behavior of a very special wave. It starts out looking like a single wave, but it is actually composed of 2 solitons, that is, 2 waves that tend to maintain their own identity. These waves each have a characteristic shape and velocity. They can pass through each other, temporarily making what looks like a single wave, but they then separate again. In this simulation, we are watching the waves separate.

For special initial conditions, you get this nice picture. If you want to see 3 solitons, look at the code and change the appropriate number. If the initial condition does not have the right value, you'll get a more complicated wave pattern. Note also that when the waves begin to interact with the boundary, we get jaggy lines. This is because the soliton solution really only works on an infinite domain. Our region is too small, and our treatment of the boundary conditions too simple, to avoid this problem.

**Assignment**

**COMPUTATION**: Some nonlinear BVP's have multiple solutions. Solve the chemical BVP using the shooting method. Use the value **lambda = - 1**. Record the value of **ALPHA** that the shooting method returns. There are actually **two** different solutions. To find the second one, you must vary the starting values that you give to the secant method. I will tell you that each pair of starting values for the secant method can be of the form **(N,N+1)** where **N** is an integer between 0 and 15.

Secant starting values ALPHA Y(1.0)  
 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_