MATH2071: LAB 4: BVPs and PDEs

1 Introduction

The initial value problem for ordinary differential equations of the previous labs is only one of the two major types of problem for ordinary differential equations. The other type is known as the "boundary value problem" (BVP). A simple example of such a problem would describe the shape of a rope 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?

This lab is concerned with two of the most common approaches to solving BVPs as well as a combined IVP-BVP for a partial differential equation. The extra credit problem introduces a third approach to solving BVPs. The discussion in this lab is limited to relatively simple approaches in a single space dimension and is intended to give the flavor of these approaches, each of which could easily be the subject of a full semester's course. Except for the extra credit exercise, these methods are easily extended to two and three space dimensions.

The approaches included in this lab are the following:

- The Finite Difference method (FDM),
- The Finite Element method (FEM),
- The Method of Lines, and,
- The Shooting method (extra credit).

This lab will take four sessions. If you print this lab, you may prefer to use the pdf version.

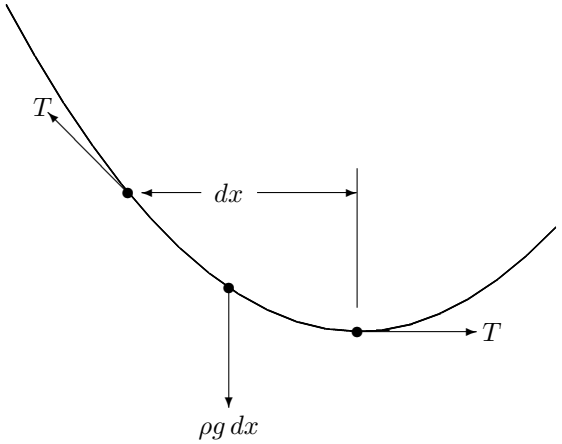
2 Boundary Value Problems

A one-dimensional boundary value problem (BVP), is similar to an initial value problem, except that the data we are given isn't conveniently located at a starting point, 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).

We will be using the following problem as the illustrative example for several of the following exercises. **The clothesline BVP**: A rope is stretched between two points. If the rope were weightless, or if it were 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. We wish to determine the curve described by the rope. We will use the variable x to denote horizontal distance and $u(x)$ to denote height of the rope at the point x.

Forces on a clothesline



The equation for the curve described by the rope can be derived (this is not a proof: it is a description of why you should believe the equation) in the following manner. Suppose that the tension in the rope is T (a constant because the rope is in equilibrium), and consider a tiny piece of the rope of length dx, and with mass per unit length ρ. The total mass of the differential piece of rope is $\rho\, dx$, so that the force due to gravity is directed downward and is given by $\rho g\, dx$. This piece of rope observes forces on each of its ends. The magnitudes of these forces are equal to the tension, T, and the directions are given by the slope of the curve at the ends of the differential piece. Hooke's law says that the tension is proportional to the amount of strain in the string, $T = -K du/dx$, where K is a constant of proportionality (Young's modulus). Hence, the equation can be written as

$$-K\,\frac{du}{dx}\Big]_{\text{right}} + K\,\frac{du}{dx}\Big]_{\text{left}} = -\rho g\, dx.$$

Dividing both sides by dx and letting $dx \to 0$ yields the equation

$$-\frac{d}{dx}\left(K\frac{du}{dx}\right) = -\rho g$$

. There is no reason that the "constant" of proportionality cannot change from place to place. For the sake of definiteness, assume K varies as $K(x) = 1 + cx$, for constant C, representing a rope (or spring) whose stiffness varies from end to end. The result is the equation

$$(1 + cx)u'' + cy' = \rho g.$$

When $c = 0$, this equation is called the "Poisson equation" and also describes the distribution of heat in a solid bar, among other common physical problems.

For the sake of definiteness, take $\rho g = 0.4$ and $c = 0.05$, the left end of height 1 at $x = 0$, and the right end height of 1.5 at $x = 5$. Thus, the system to be solved is

$$\begin{aligned} (1 + cx)u'' + cu' &= \rho g \\ c &= 0.05 \\ \rho g &= 0.4 \\ u(0) &= 1 \\ u(5) &= 1.5 \end{aligned} \tag{1}$$

The ODE is *linear*. Linearity implies good things such as the existence and uniqueness of solutions.

3 Finite Difference Method for a BVP

The "derivation" presented above for the shape of the rope is suggestive of a way to solve for the shape, called the "finite difference method." Assume that we have divided the interval up into N equal intervals of width Δx determined by $N+1$ points. Denote the spatial points x_n, $n = 0, 1, \ldots, N+1$. Approximate the value of $y(x_n)$ by y_n. Also approximate the Young's modulus function as $K_n = K(x_n) = (1 + cx_n)$.

Now, consider the n^{th} interval as if it were the differential piece of rope mentioned in the derivation. Using the standard finite difference approximation for a derivative, the slope of the rope at the left of the n^{th} interval could be approximated as $(y_n - y_{n-1})/\Delta x$ and the slope on the right of the n^{th} interval could be approximated as $(y_{n+1} - y_n)/\Delta x$. The difference between these is an approximation of the second derivative

$$y_n'' \approx \frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta x^2}.$$

Along similar lines, approximate the first derivative as

$$y_n' \approx \frac{y_{n+1} - y_{n-1}}{2\Delta x}.$$

Both approximations (of y' and y'') have the same Taylor-series (truncation) error of $O(\Delta x^2)$.

Put all these together into (1) to get

$$(1 + cx_n)\frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta x^2} + c\frac{y_{n+1} - y_{n-1}}{2\Delta x} = \rho g$$

and, as above, $c = 0.05$ and $\rho g = 0.4$. We can associate this equation with the solution value at y_n, except for $n = 0$ and $n = N + 1$ (do you see why?). Conveniently, those are 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 1, 2, 3, and 4, and associate the boundary conditions with the $n = 0$ and $n = 5$ solution values. Note that $x_n = n\Delta x$. I also multiplied through by Δx^2 to make things look nicer:

$$\begin{aligned}
u_0 &= 1 \\
(1 + c\Delta x - c\Delta x/2)u_0 - 2(1 + c\Delta x)u_1 + (1 + c\Delta x + c\Delta x/2)u_2 &= 0.4\Delta x^2 \\
(1 + 2c\Delta x - c\Delta x/2)u_1 - 2(1 + 2c\Delta x)u_2 + (1 + 2c\Delta x + c\Delta x/2)u_3 &= 0.4\Delta x^2 \\
(1 + 3c\Delta x - c\Delta x/2)u_2 - 2(1 + 3c\Delta x)u_3 + (1 + 3c\Delta x + c\Delta x/2)u_4 &= 0.4\Delta x^2 \\
(1 + 4c\Delta x - c\Delta x/2)u_3 - 2(1 + 4c\Delta x)u_4 + (1 + 4c\Delta x + c\Delta x/2)u_5 &= 0.4\Delta x^2 \\
u_5 &= 1.5
\end{aligned} \tag{2}$$

Actually, in Equation (2), the quantities u_0 and u_5 are not really variables, being fixed by the boundary conditions. Hence the only variables are u_1, u_2, u_3 and u_4. The system can be rewritten as

$$\begin{aligned}
-2(1 + c\Delta x)u_1 + (1 + 1.5c\Delta x)u_2 + 0 + 0 &= 0.4\Delta x^2 - (1 + 0.5c\Delta x)u_0 \\
(1 + 1.5c\Delta x)u_1 - 2(1 + 2c\Delta x)u_2 + (1 + 2.5c\Delta x)u_3 + 0 &= 0.4\Delta x^2 \\
0 + (1 + 2.5c\Delta x)u_2 - 2(1 + 3c\Delta x)u_3 + (1 + 3.5c\Delta x)u_4 &= 0.4\Delta x^2 \\
0 + 0 + (1 + 3.5c\Delta x)u_3 - 2(1 + 4c\Delta x)u_4 &= 0.4\Delta x^2 - (1 + 4.5c\Delta x)u_5
\end{aligned}$$

and this system has been formatted to suggest the matrix equation

$$\begin{bmatrix}
-2(1 + c\Delta x) & +(1 + 1.5c\Delta x) & +0 & +0 \\
(1 + 1.5c\Delta x) & -2(1 + 2c\Delta x) & (1 + 2.5c\Delta x) & +0 \\
0 & +(1 + 2.5c\Delta x) & -2(1 + 3c\Delta x) & +(1 + 3.5c\Delta x) \\
0 & +0 & +(1 + 3.5c\Delta x) & -2(1 + 4c\Delta x)
\end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} =
\begin{bmatrix}
0.4\Delta x^2 - (1 + 0.5c\Delta x)u_0 \\
0.4\Delta x^2 \\
0.4\Delta x^2 \\
0.4\Delta x^2 - (1 + 4.5c\Delta x)u_5
\end{bmatrix} \tag{3}$$

By discretizing the differential equations we have created a set of linear algebraic equations that have the symbolic form $AU = b$. To set up and solve the equations (3) in Matlab, we could type:

```
N = 4;
C = 0.05;
RHOG = 0.4;
% N interior mesh points, N+1 intervals
dx = 5.0 / ( N + 1 );
x = dx * (0:N+1);
A = [ -2*(1+C*dx)    +(1+1.5*C*dx)        0              0;
      +(1+1.5*C*dx)  -2*(1+2*C*dx)  +(1+2.5*C*dx)        0;
            0        +(1+2.5*C*dx)  -2*(1+3*C*dx)  (1+3.5*C*dx);
            0              0        +(1+3.5*C*dx) -2*(1+4*C*dx) ];
ULeft=1;
URight=1.5;
b = [ RHOG*dx^2-(1+0.5*C*dx)*ULeft
      RHOG*dx^2
      RHOG*dx^2
      RHOG*dx^2-(1+4.5*C*dx)*URight];
U = A \ b;
U = [ULeft; U; URight]
```

Make sure you understand the first and last components in b. You should recall that the backslash notation is shorthand for saying `U=inv(A)*b` but tells Matlab to solve the equation `A*U=b` without actually forming the inverse of `A`.
Remark: The vector `x` is a row vector and the vector `U` is a column vector! This is the convention that has been followed for the `_ode.m` files and will be followed throughout these labs.

Exercise 1: In this exercise, you will be using the above code to solve the rope BVP. You will also be exhaustively checking that the code is correct.

(a) Copy the above code and paste it into a script m-file named `exer1a.m`. Execute `exer1a` to find a solution of Equation (3). Please include the printed values of `U` as part of the lab summary.

(b) Verify that the values of `U` and `b` that you found satisfy at least one of the middle four equations in (2). To do this, write a script m-file named `exer1b.m` and plug the values of c, Δx, u_k, $k = 0, \ldots, 5$ into your chosen equation. Show the result is essentially zero.
Be careful! Lower-case c is upper-case `C`, Δx is `dx` and u_k, $k = 0, 1, \ldots 5$ is `U(1:6)` in `exer1a`.

(c) Direct substitution into (3) shows that the function $u(x) = 1$ would be a solution if $\rho g = 0$ and $u(5) = 1$. As a second verification step, make a copy of `exer1a.m` called `exer1c.m` with `rhog=0` and `URight=1` and check that the discrete solution is (exactly or to roundoff) correct.

(d) Direct substitution into (3) shows that the function $u(x) = x$ would be a solution if $\rho g = c$, $u(1) = 0$ and $u(5) = 5$. As a third verification step, make a copy of `exer1a.m` called `exer1d.m` with `rhog=C` and and with boundary values `ULeft` and `URight` chosen to match the solution you are testing. Check that the discrete solution is (exactly or to roundoff) correct.

(e) Direct substitution into (3) shows that the function $u(x) = x^2$ would be a solution if $\rho g = 2 + 4cx$, $u(1) = 0$) and $u(5) = 25$. As a fourth verification step, make a copy of `exer1a.m` called `exer1e.m` with constant `rhog`, which is used four times, replaced by the four values of the *vector* `2+4*c*x(2:5)'` and with boundary values chosen to agree with $u(x) = x^2$ and check that the discrete solution is (exactly or to roundoff) correct.

(f) Now that you are confident that the code is correct, use `exer1a.m` to solve the unmodified BVP (3). Plot `U` versus `x`. It should appear roughly parabolic, like a rope hanging from its ends, and pass through $(0, 1)$ and $(5, 1.5)$ on its ends. Please include this plot with your summary.

Remark: The exact solutions $u = 1$, $u = x$, and $u = x^2$ can be used as exact *discrete* solutions and verification tests *only* when the approximation expressions for first and second derivatives are sufficiently accurate. Because the mesh is uniform, the expressions used here give rise to truncation error of $O(\Delta x^3)$ so that polynomials up to quadratic will be exactly represented.

The purpose of the previous exercise is to verify that you have copied the code correctly and to illustrate the powerful verification strategy of checking against known *exact discrete* solutions. You should always use a small, simple problem to verify code by comparison with hand calculations. If possible, you should also compare results with theoretical results and with results achieved using a different method. In the next exercise you will be modifying the above code to handle the case of large `N` and solving a slightly more realistic problem. Of course, you don't want to bother typing in the matrix `A` if `N` is 100 or more, so you will be writing Matlab code to do it.

Exercise 2:

(a) Make a copy of the script m-file `exer1a.m` and change it into a function m-file called `rope_bvp.m` with the signature

```
function [x,U] = rope_bvp(N)
% [x,U] = rope_bvp(N)
% comments

% your name and the date
```

(b) Add comments after the signature line and modify the matrix (`A`) and right side vector (`b`) generation statements to be valid for arbitrary values of `N`. Make sure that the vector `U` is a column vector. (You should also eliminate the line `N = 4`.) **Hint**: You can use the `zeros(N,N)` statement to generate an `N`-by-`N` matrix of all zeros for `A` and then fill in the non-zero values. You can use the command `ones(N,1)` to construct a column vector of length `N` containing all ones.

(c) Check your work by running `rope_bvp` for `N=4` and confirming that you get the same values of `U` as from `exer1a.m`. One easy way to do this is to first run `exer1a`, then use the command `[x1,U1]=rope_bvp(4)` and check that `U-U1` is the zero vector.
Debugging: If the results are not correct, print the matrix `A` from `rope_bvp` and check it against the matrix `A` from `exer1a`. Do the same for the vectors `b`. *Fix any mistakes before continuing.*

(d) You may still have mistakes in the treatment of `N`. To be sure of your code, make a copy of `rope_bvp.m` and modify it to get the constant solution $u = 1$ as you did for `exer1c.m` above. Check your results for `N=5`. *Fix any mistakes before continuing.*
If you cannot find your mistakes by checking your code, re-do Equations (2) for $n = 6$ and check the terms against your code, one term at a time.

(e) As a second test, make a copy of `rope_bvp.m` and modify it to get the linear solution $u = x$ as you did for `exer1d.m` above. Check your results for `N=5`. *Fix any mistakes before continuing.*
If you cannot find your mistakes by checking your code, re-do Equations (2) for $n = 6$ and check the terms against your code, one term at a time.

(f) As a third test, make a copy of `rope_bvp.m` and modify it to get the quadratic solution $u = x^2$ as you did for `exer1d.m` above. Check your results for `N=5`. *Fix any mistakes before continuing.*
If you cannot find your mistakes by checking your code, re-do Equations (2) for $n = 6$ and check the terms against your code, one term at a time.

(g) Now we are ready to solve the big problem! Use `N=119` so that there are 119 unknowns `U(1:119)`. Call the new solution `[x2,U2]`, and plot `U2` versus `x2`. Re-run `exer1a` to get `U` and `x`, and plot `U` versus `x` as circles (`plot(x,U,'o')` on the same frame (`hold on`) and send the single frame with both plots to me with the summary. To help in grading, please include the value of `U(50)` in your summary.

Remark: You may wonder why the four-point mesh solution and the 119-point mesh solution seem to agree at the four common points. This behavior is highly unusual. It happens because the solution of the differential equation is almost quadratic and the difference scheme exactly reproduces quadratic functions. For larger values of c, the solution looks less like a quadratic, and the solution for $N = 4$ agrees less well with the solution for $N = 119$. Try it, if you wish.

4 Finite element method

In the previous section, you saw an example of the finite difference method of discretizing a boundary value problem. This method is based on a finite difference expression for the derivatives that appear in the equation itself. The finite difference method results in a list of values that approximate the true solution at the set of mesh points. Approximate values between the mesh points might be generated using interpolation ideas, but the method itself does not depend on any such interpolation. The reason that $N = 119$ was chosen for comparison with $N = 4$ in the previous exercise is because the four x values in the $N = 4$ case appear among the 119 x-values in the $N = 119$ case.

An alternative approach, called the "finite element method" (FEM) is based on approximating the unknown as a sum of simple "shape functions" defined over the mesh intervals. Since the finite element solution is actually a function, it is defined over the same spatial interval as the true solution and much of the machinery of functional analysis is available for proving facts about the method and solutions that arise. As a consequence, the FEM occupies a large part of the mathematics literature. You can find the FEM discussed in Quarteroni, Sacco, and Saleri, Sections 12.4 and 12.5.

In this section, you will see the FEM applied to a particular boundary value problem. The problem is somewhat simpler than the clothesline problem discussed above, but contains the same essential features. Consider the equation

$$y'' + y' + y = f(x) \tag{4}$$

defined for x in the interval $[0, 1]$, for $f(x)$ a given function, and with boundary values

$$y(0) = y(1) = 0. \tag{5}$$

While the finite difference method attacks (4) directly, the FEM starts from the so-called "weak" form of the equation. This form can be constructed from (4) by multiplying through by a function $v(x)$, assumed to satisfy the same boundary conditions (5), and integrating some terms by parts. In this case, the weak form is given by

$$-\int_0^1 y'(x)v'(x)dx + \Big[y'(x)v(x)\Big]_0^1 + \int_0^1 y'(x)v(x)dx + \int_0^1 y(x)v(x)dx = \int_0^1 f(x)v(x)dx$$

Since $v(x)$ satisfies (5), the bracketed term drops out and the result is

$$-\int_0^1 y'(x)v'(x)dx + \int_0^1 y'(x)v(x)dx + \int_0^1 y(x)v(x)dx = \int_0^1 f(x)v(x)dx. \tag{6}$$

Remark: In this case, only the first term has been integrated by parts. Some authors might also integrate the second term by parts. Doing so would not change the following discussion very much.

To approximate the function y, choose an odd integer N and a set of functions $\phi_n(x)$ for $n = 1, 2, \ldots, N$, defined on the interval $[0, 1]$, that form a basis of some reasonable approximating function space. For most finite element constructions, these functions satisfy the following characteristics:

1. They are continuous and piecewise polynomials.

2. Each of the functions takes the value 1 at a single mesh node and zero at all other mesh nodes.

In this exercise, the functions will be piecewise quadratic polynomials and the mesh nodes are given by dividing the interval into $N+1$ subintervals, each of length $h = 1/(N+1)$, so that a sequence of spatial points is given by $x_n = nh$ for $n = 0, 1, \ldots, N, N+1$ ($x_0 = 0$ and $x_{N+1} = 1$). The quadratic Lagrange functions are defined in the following way (see Quarteroni, Sacco, Saleri, p. 562).

$$(n \text{ even})\ \phi_n(x) = \begin{cases} \frac{(x-x_{n-1})(x-x_{n-2})}{(x_n-x_{n-1})(x_n-x_{n-2})} & \text{for } x_{n-2} < x \le x_n, \\ \frac{(x_{n+1}-x)(x_{n+2}-x)}{(x_{n+1}-x_n)(x_{n+2}-x_n)} & \text{for } x_n < x \le x_{n+2} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$(n \text{ odd})\ \phi_n(x) = \begin{cases} \frac{(x_{n+1}-x)(x-x_{n-1})}{(x_{n+1}-x_n)(x_n-x_{n-1})} & \text{for } x_{n-1} \le x \le x_{n+1} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This collection of functions is known to form a basis for a function space that includes all constant, linear, and quadratic functions on $[0, 1]$ and it has good approximation properties. It is also true that each of the functions $\phi_n(x)$, for $n = 1, \ldots, N$ satisfies the boundary conditions (5).

Assume that an approximate solution to (6) can be written as

$$y(x) = \sum_{n=1}^{N} y_n \phi_n(x) \tag{9}$$

for (as yet unknown) constants y_n. Plugging (9) into (6) and choosing $v(x) = \phi_m(x)$ yields N equations of the form

$$\sum_{n=1}^{N} \underbrace{\left(-\int_0^1 \phi_n'(x)\phi_m'(x)dx + \int_0^1 \phi_n'(x)\phi_m(x)dx + \int_0^1 \phi_n(x)\phi_m(x)dx \right)}_{a_{mn}} y_n = \underbrace{\int_0^1 f(x)\phi_m(x)dx}_{f_m}. \tag{10}$$

Regarding the values y_n as the components of a (column) vector $\mathbf{Y}$, the values a_{mn} as the components of a matrix $\mathbf{A}$, and the values f_m as the components of a (column) vector $\mathbf{F}$, then (10) can be written as the matrix equation

$$\mathbf{AY} = \mathbf{F}. \tag{11}$$

Solving the matrix equation (11) completes construction of the approximate solution (9).

In the following exercises, you will write Matlab functions to construct the basis functions $\phi_n(x)$ in (7) and (8), to evaluate the matrix elements a_{mn} and vector components f_n in (10), and solve the matrix equation (11).

Remark: When the FEM is programmed, the integrals in (11) are typically performed element-by-element. This is particularly important in multidimensional cases. Nontheless, we will be using a conceptually simpler approach to the integrations.

Exercise 3: In this exercise, you will construct the Lagrange quadratic basis functions. The values x_k used in (7) and (8) will be evaluated as $x_k = kh$, where $h = 1/(N+1)$. The expression kh is valid even when $k = 0$, although Matlab does not allow subscripts equal to zero.

In the Matlab functions below, you should regard the variable `x` as a *scalar* value, not a vector. Attempting to write vector (componentwise) code only complicates matters here.

(a) Write a Matlab function m-file for $\phi_n(x)$ by completing the following outline

```
function z=phi(n,h,x)
% z=phi(n,h,x)
% Lagrange quadratic basis functions
```

```
% your name and the date

if numel(x) > 1
  error('x is a scalar, not a vector, in phi.m');
end

if mod(n,2)==0  % n is even

  if (n-2)*h < x & x <= n*h
    z= ??? code implementing first part of (7) ???
  elseif n*h < x & x <= (n+2)*h
    z= ??? code implementing second part of (7) ???
  else
    z=0;
  end

else  % n is odd

  ??? code implementing (8) ???

end
```

(b) Plot some of your functions using the following code

```
N=7;
h=1/(N+1);
x=linspace(0,1,97);
mesh=linspace(0,1,N+2);
for k=1:numel(x)
  y3(k)=phi(3,h,x(k));
  y4(k)=phi(4,h,x(k));
end
plot(x,y3,'b')
hold on
plot(x,y4,'r')
plot(mesh,zeros(size(mesh)),'*')
hold off
```

You should observe that each ϕ takes the value 1 at a *single* mesh node (`x(k)`, indicated with an asterisk), takes the value zero at all other mesh nodes, is continuous, and is parabolic or zero between any two mesh nodes. Please include this plot with your summary file.

(c) Examine the definitions (7) and (8) and show that $\phi_n(x)$ is a continuous function by showing that the pieces match up at x_{n-2}, x_n, and x_{n+2} for even n and at x_{n-1} and x_{n+1} for odd n. (You don't need Matlab to do this.) Similarly, show that

$$\phi_n(x_m) = \begin{cases} 1 & m = n \\ 0 & \text{otherwise} \end{cases}$$

(d) Write a Matlab function m-file similar to `phi.m` for the derivative $\phi_n'(x)$. Differentiate (7) and (8) by hand to find $\phi_n'(x)$ and use your formulæ for the function `phip.m` with signature

```
function z=phip(n,h,x)
```

```
% z=phip(n,h,x)
% derivative of Lagrange quadratic basis functions

% your name and the date
```

(e) For the case N=7, plot $\phi_3(x)$ and $h\phi_3'(x)$ (multiply by h to get a better scaling) on the same plot. Examine the plot carefully and convince yourself that ϕ_3' appears to be the derivative of ϕ_3.

(f) Similarly, plot ϕ_4 and ϕ_4'. Examine both cases from (7).

(g) For the case N=7 and the point $x = 0.4$, use the finite difference expression

$$\frac{\phi_3(x + \Delta x) - \phi_3(x - \Delta x)}{2\Delta x}$$

with $\Delta x = 0.01$ to estimate $\phi_3'(x)$. Does it agree up to roundoff with the result from `phip`? Similarly for ϕ_4.

In the following three exercises, you will write m-files to construct and verify the three pieces of the matrix A in (10). Following that, you will construct the full matrix A and solve for the finite element solution of the given problem (4).

Exercise 4: In this exercise, you will generate the first part of the matrix **A**,

$$a_{mn}^{(1)} = -\int_0^1 \phi_m'(x)\phi_n'(x)dx \tag{12}$$

You will be using many of the mathematical facts about this quantity in order to check that your code is correct.

In order to do the integrations, you will be using code that I give you that provides a uniform way to do the required integrations.

(a) Download a copy of a special integration function `gaussquad.m`. This code takes the names of two functions (such as `'phi'` or `'phip'`) along with their appropriate subscripts and the value of the mesh spacing `h` and integrates the resulting product over the interval `[a,b]`. Its signature is

```
function q=gaussquad(f1,k1,f2,k2,h,a,b)
```

This `gaussquad` function will provide the *exact* value, not merely an approximate value, of the integral for the cases considered in this lab: piecewise low degree polynomials.

(b) Choose `N=7` and `h=1/(N+1)`, and write an m-file named `exer4.m` to compute the matrix values $a_{mn}^{(1)}$ in (12). Call the resulting matrix `A1`. Do not overlook the fact that the basis function derivatives appear in (12), not the basis functions themselves, and there is that pesky minus sign in front of the integral. Please include the values of `A1` in your summary file.
Remark 0: The following two remarks concern program efficiency. Real programs intended to solve large problems should be concerned with efficiency, but the first time you write a program you should strive for simplicity and clarity. It is easier to make a correct program run fast than it is to make a fast program run correctly.
Remark 1: The support of $\phi_n(x)$ is contained in the interval $[(n-2)h, (n+2)h]$. You could use this fact to shrink your integration limits and improve the efficiency of the integration, but it is not required.
Remark 2: Because the support of $\phi_n(x)$ and of $\phi_m(x)$ do not intersect when $|n - m| \geq 4$, you can take advantage of this fact to avoid computing components $a_{mn}^{(1)}$ that must be zero, but it is not required.

(c) (12) indicates that the matrix `A1` is symmetric. Check that your computation is symmetric by showing that

```
norm(A1-A1','fro')
```

is zero or roundoff. Add this code to `exer4.m`.

(d) For the case `N=7` and `h=1/(N+1)`, add code to `exer4.m` to compute the function $\psi(x) = \sum_{n=1}^{N} \phi_n(x)$ for the 97 values `x=linspace(0,1,97)` and plot it. You should observe that it is equal to 1 except near the endpoints of the interval. As a consequence, it has zero derivative, except near the endpoints of the interval. Please include the plot with your summary.

(e) Since `A1*ones(N,1)`$= \sum_{n=1}^{N} \int_0^1 \phi_n'(x)\phi_m'(x)dx = \int_0^1 \psi'(x)\phi_m'(x)dx$, and you just saw that ψ' is zero except for $x_0 \le x \le x_2$ and $x_6 \le x \le x_8$, verify the zeros in positions $m = 3, 4, 5$.

(f) Noting that $a_{mn}^{(1)} = \int_0^1 \phi_m(x)\phi_n''(x)dx$, you should be able to see why the following code

```
N=7;
h=1/(N+1);
v=(1:N)'*h; % v=x
A1*v
```

should yield a vector that is zero except in the positions $m = 6, 7$. Add this code to `exer4.m` and verify that it does.

(g) Recall that the BVP $-y'' = 2$ with $y(0) = y(1) = 0$ has solution $x(1-x)$. You can solve this BVP using FEM. Write a Matlab function m-file with the signature

```
function z=rhs4(n,h,x)
% z=rhs4(n,h,x)

% your name and the date
```

to compute the constant function equal to (-2) everywhere. (This is an almost trivial exercise. It is needed so that `gaussquad.m` can be used.) Add code to `exer4.m` to use the `gaussquad.m` function to compute the vector components $(f_4)_m = \int_0^1 f_4(x)\phi_m(x)dx$ and call the resulting vector `RHS4`. Since the quadratic function $x(1-x)$ satisfies the boundary conditions, it can be written *exactly* as a combination of the ϕ_n! Hence, the following code should yield the zero vector.

```
N=7;
h=1/(N+1);
xx=(1:N)'*h;   % the variable x has already been used.
v=xx.*(1-xx);
v-A1\RHS4  % should be zero
```

Add this code to `exer4.m`. Please include the values of `RHS4` in your summary.

The tests in `exer4.m` construct and test the matrix `A1`, so you should now be reasonably sure that `A1` is correct. In the following exercise, you will construct and test `A3`. In the subsequent exercise, you will construct and test `A2` so that you have all of the matrix `A`.

Exercise 5:

(a) Start writing an m-file named `exer5.m` similar to `exer4.m` but for the terms

$$a_{mn}^{(3)} = \int_0^1 \phi_m(x)\phi_n(x)dx.$$

Call the resulting matrix `A3`. Include this matrix in your summary.
Warning: If you copy code from `A1` for `A3`, don't forget that `A1` has a minus sign in it from the integration by parts but that `A3` does not.

(b) Add code to check that `A3` is symmetric.

(c) Note that the boundary value problem

$$y'' + y = -2 + x(1-x) \tag{13}$$

with $y(0) = y(1) = 0$ has the exact solution $y = x(1-x)$, and this quadratic function can be expressed exactly as a sum of the ϕ_n.

(d) Write a Matlab function m-file `rhs2.m` with signature

```
function z=rhs5(k,h,x)
% z=rhs5(k,h,x)

% your name and the date
```

to compute the function $f_5(x) = -2 + x(1-x)$. Add code to `exer5.m` to use `rhs5.m` and `gaussquad.m` to compute the (column) vector $\int_0^1 f_5(x)\phi_m(x)dx$, and call the resulting vector `RHS5`. Include these values in your summary

(e) The matrix `A1` was computed by `exer4.m`, and `A2` and `RHS5` are computed by `exer5.m`. Add code to `exer5.m` to solve the matrix equation `(A1+A3)*Y=RHS5`. You have just solved (13), and your solution should equal $x_n(1-x_n)$ at each of the nodes $x_n = nh$ *exactly*, with only roundoff errors. Add code to `exer5.m` to check that this is true. If it is not true, there is a mistake somewhere. Fix it before continuing.

Exercise 6:

(a) Write another m-file named `exer6.m` to compute the terms

$$a^{(2)}_{mn} = \int_0^1 \phi_m(x)\phi'_n(x)dx \tag{14}$$

Call this matrix `A2`. Include this matrix in your summary.

(b) Integrating (14) by parts and applying the boundary conditions shows that the matrix `A2` is skew-symmetric (*i.e.,* `A2'=-A2`. Add code to `exer6.m` to confirm this is true.

(c) Note that adding the terms `A1`, `A2` and `A3` together generates the matrix `A`.

(d) Note that boundary value problem

$$y'' + y' + y = -2 + (1-2x) + x(1-x) \tag{15}$$

with boundary values $y(0) = y(1) = 0$ has exact solution $y = x(1-x)$.

(e) Write a Matlab function m-file `rhs6.m` with signature

```
function z=rhs6(k,h,x)
% z=rhs6(k,h,x)

% your name and the date
```

to compute the function $f_6(x) = -2 + (1-2x) + x(1-x)$. Add code to `exer6.m` to compute the (column) vector $\int_0^1 f_6(x)\phi_m(x)dx$, and call the resulting vector `RHS6`.

(f) Solve the matrix equation `A*Y=RHS6`. You have just solved (15), and your solution should equal $x(1-x)$ *exactly*, with only roundoff errors.

Remark: Writing test scripts like `exer4.m`, `exer5.m` and `exer6.m` allows you to re-test your work at any time. This strategem helps you maintain confidence in your code and also allows you to propose and test modifications easily.

Exercise 7: The boundary value problem

$$y'' + y' + y = x + 1$$

with boundary values $y(0) = y(1) = 0$ has exact solution $y = x - e^{-0.5(x-1)} \sin \omega x / \sin \omega$, with $\omega = \sqrt{3}/2$.

(a) Write a function m-file `exact7.m` to evaluate the above exact solution.

(b) Write a function m-file `rhs7.m` to evaluate the right side function $z = x + 1$.

(c) Write a function m-file `solve7.m` with signature

```
function [x,Y]=solve7(N)
% [x,Y]=solve7(N)
%    ... more comments ...

% your name and the date
```

that performs the following tasks:

i. Compute the finite element matrix `A`

ii. Compute the right side vector `RHS7`

iii. Solve the system `A*Y=RHS7` for `Y`

iv. Compute the spatial coordinate vector `(1:N)'*h`

(d) Fill in the following table and estimate the rate of convergence of this method. Measure the error as the maximum absolute value of the difference between the calculated and true solutions at the nodes $x_n = nh$, and take the ratio as the error(h) divided by error(h/2).

N	h	error	ratio
7	1.2500e-1	________	________
15	6.2500e-2	________	________
31	3.1250e-2	________	________
61	1.6129e-2	________	________
121	8.1967e-3	________	

Remark: The rate of convergence is higher than expected from theory because the mesh is uniform.

5 Burgers' Equation

A partial differential equation (PDE) involves derivatives of a function u which depends on more than one independent variable. One interesting PDE is the one-dimensional *Burgers' equation*, which is a one-dimensional nonlinear equation whose nonlinear term is similar to the one in the Navier-Stokes equations of fluid flow. In this case, the variable will be called u and it is a function of space (x) and time (t), $u(x,t)$. The variable u is called "velocity" in the Navier-Stokes equations. Burgers' equation on the spatial interval $[0,1]$ can be written as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{16}$$

where ν is a constant. Boundary conditions can be taken as $u(0,t) = 1$ and $u(1,t) = 0$, both for all time. Two space boundary conditions are necessary because the equation is second-order in space. Since the equation is first-order in time, only one intitial condition is needed.

To get an idea of what the solution to Burgers' equation might look like, first imagine that $\nu = 0$ and also that the coefficient $u = v$ is a constant, so that the equation becomes the wave equation

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0.$$

If $v > 0$, this equation represents a right-going wave that moves without changing shape. To see this, note that for any given function of one variable, f, $u = f(tv - x)$ is a solution of the wave equation. Changing ν to a small, positive number means that the wave propagates as before, but slowly spreads and decays to zero. Finally, since the coefficient v is *not* constant, the wave propagates faster where u is larger and slower where u is smaller. Thus a wave that is larger to the left and smaller to the right will steepen as it propagates to the right.

6 The Method of Lines

Look at (16) and pretend, just for a moment, that time is frozen. The equation suddenly starts looking like the boundary value problem (1) that we solved before, only with a extra term on the left and with $\partial u/\partial t$ replacing the right side. This observation is the basis for the "method of lines," wherein the spatial discretization is performed separately from the temporal. Consider the function $u(t, x)$, but think of it as a function of x first. The resulting BVP can be written with primes denoting spatial differentiation so it looks more like what we have been doing.

$$\nu u'' - uu' = \frac{\partial u}{\partial t},$$

where we are focussing on the left side for the moment, along with spatial boundary conditions $u(0, t) = 1$ and $u(1, t) = 0$.

We solved a BVP a lot like this one above. We broke the interval $[0, 1]$ into $\mathtt{N}+1$ subintervals and labelled the $(\mathtt{N}+2)$ resulting points $x_n, n = 0, 1, 2, ..., (\mathtt{N}+1)$. Next, we defined u_n as being the approximate solution at x_n. Keep in the back of you mind, though, that u_n is really still a function of t, $u_n(t)$. We will denote the vector of values u_n by U, because we have already used the unsubscripted u to denote the continuous solution. Keep in mind that $U(t)$ is a function of time. We will use the same finite difference discretization as before for the term u''

$$u'' \approx \frac{u_{n+1} - 2u_n + u_{n-1}}{\Delta x^2} \tag{17}$$

and will choose a natural discretization for the first-order nonlinear term

$$uu' \approx u_n \frac{u_{n+1} - u_{n-1}}{2\Delta x} \tag{18}$$

(The truncation error of each of these forms is $O(\Delta x^2)$.) The resulting discrete equations become

$$\frac{\partial u}{\partial t} = \nu \frac{u_{n+1} - 2u_n + u_{n-1}}{\Delta x^2} - u_n \frac{u_{n+1} - u_{n-1}}{2\Delta x}. \tag{19}$$

But we remain on familiar ground: (19) is just a system of IVPs! We know a bunch of different methods to solve it. It turns out that the system is moderately stiff, and will become more stiff when `N` is taken larger and larger. We will use backwards Euler to solve this system. Recall that backwards Euler requires an m-file to evaluate both the function $\mathbf{F}$ and its partial derivative (Jacobian)

$$\mathbf{D}_{mn} = \frac{\partial \mathbf{F}_m}{\partial \mathbf{U}_n} \tag{20}$$

In Matlab notation, the variable U will be a matrix whose entries U_{kn} approximate the values $u_n(t_k)$. In Matlab notation, for the k^{th} time interval, `U(:,k)` represents the (column vector of) values at the locations `x(:)`. The initial condition for `U` should be a column vector whose values are specified at the locations `x(:)`.

Exercise 8: In this exercise you will write, debug, and solve an m-file for the solution of Burgers' equation in the form of (19) with $\nu = 0.001$ and using `N=500`.

(a) Begin a function m-file called `burgers_ode.m` to construct the spatial discretization of Burgers' equation and its gradient. The function $\mathbf{F}$ refers to the right side of (19). Here is an outline:

```
function [F,D]=burgers_ode(t,U)
% [F,D]=burgers_ode(t,U)
% compute the right side of the time-dependent ODE arising from
% a method of lines reduction of Burgers' equation
% and its derivative, D.
% Boundary conditions are fixed =0 at the
% endpoints x=0 and x=1.
% A fixed number of spatial points (=N) is used.
% The variable t is not used, but is kept as a place holder.
% U is the vector of the approximate solution at all spatial points
% output F is the time derivative of U (column vector)
% output D is the Jacobian matrix of
%   partial derivatives of F with respect to U

% spatial intervals
N=500;
NU=0.001;
dx=1/(N+1);
ULeft=1;  % left boundary value
URight=0; % right boundary value

F=zeros(N,1);  % force F to be a column vector
D=zeros(N,N); % matrix of partial derivatives

% construct F and D in a loop
for n=1:N
  if n==1  % left boundary
     F(n)  = ??? Function, left endpoint ???
     D(?,?)= ??? Derivative, left endpoint ???
  elseif n<N  % interior of interval
     F(n)  = ??? Function, interior points ???
     D(?,?)= ??? Derivative, interior points ???
  else % right boundary
     F(n)  = ??? Function, right endpoint???
     D(?,?)= ??? Derivative, right endpoint???
  end
end
```

(b) It is easiest to treat the ***interior*** points (the case `n<N`) first. Replace the line

```
F(n)= ??? Function, interior points ???
```

with the discretization for `F(n)` given in (19).

(c) Replace the line

```
D(?,?)= ??? Derivative, interior points ???
```

with the values `D(n,n)`, `D(n,n+1)`, and `D(n,n-1)` according to the formula that was given above in (20) and is repeated here.

$$\mathbf{D}_{nm} = \frac{\partial \mathbf{F}_n}{\partial \mathbf{U}_m} \tag{21}$$

The variable m will take on the three values `n`, `n+1`, and `n-1`. To do, for example, `D(n,n+1)`, write out the expression for `F(n)` and differentiate it with respect to `U(n+1)`. **Remark:** It is easy to see from the formula that $\mathbf{D}_{nm} = 0$ for $m < n-1$ or $m > n+1$.

(d) When `n=1`, the variable corresponding with `n-1` is the left boundary value `ULeft`. With this in mind, replace the lines

```
F(n)  = ??? Function, left endpoint ???
D(?,?)= ??? Derivative, left endpoint ???
```

with the expressions for `F(n)`, `D(n,n)` and `D(n,n+1)`.

(e) When `n=N`, the variable corresponding with `n+1` is the right boundary value `URight`. With this in mind, replace the lines

```
F(n)  = ??? Function, right endpoint???
D(?,?)= ??? Derivative, right endpoint???
```

with the expressions for `F(n)`, `D(n,n)` and `D(n,n-1)`.

(f) The spatial values represent a uniform mesh

```
N=500;  % must agree with value inside burgers_ode.m
x=linspace(0,1,N+2);
x=x(2:N+1);
```

and we will assume an initial velocity distribution that looks like a shallowly sloped wave.

```
UInit=((1-x).^3)';
```

(g) Test your version of `burgers_ode.m` by calling it with `UInit` (the value of t does not matter) and comparing the result with the following values:

```
                results burgers_ode(0,UInit)
  n       F(n)      D(n-1,n)       D(n,n)      D(n+1,n)
  1 2.9761711475                -499.01396011  498.51296011
  2 2.9465759259    1.99800798  -499.02590030  497.02789232
250 0.0976958603  219.12262501  -501.24899902  282.12637400
499 2.395210e-05  251.00094622  -502.00194821  251.00100199
500 1.197605e-05  251.00098406  -502.00198406
```

(h) Retrieve your copies of `back_euler.m` and `newton4euler.m`, or download my copies of `back_euler.m` and `newton4euler.m`.

(i) Use `back_euler` to solve Burgers' equation starting from `UInit`. You should use 100 steps from time `t=0` to time `t=1`.

```
[t,U]=back_euler('burgers_ode',[0,1],UInit,100);
```

The solution should converge at each step. If you get the "failed to converge" message, your derivative (`D`) is probably wrong. To help with grading, please include the value of `U(200,50)` in your summary.

(j) Recall that the columns of `U` represent velocities at different *places* all at the same time and rows of `U` represent velocities at different *times* all at the same place. To see a snapshot of the solution at timestep `k`, use the command

```
plot(x, U(:,k) )
```

where `x` is the value assigned above. Please include this plot for the choice `k=50` with your summary.

(k) You can see a "flicker picture" of the evolution with the following steps

```
plot(x,U(:,1))
axis([0,1,0,1.5])
for k=2:100
  pause(0.1);
  plot(x,U(:,k));
  axis([0,1,0,1.5])
end
```

You should be able to see the "wave" steepen as it moves to the right. Please include the final frame of this sequence with your summary.

Remark: The boundary condition is inappropriate for the case that the "wave" actually reaches the right boundary, and the solution will fail if the time interval is long enough for the wave to reach $x = 1$.

7 Extra Credit: Shooting Methods (8 points)

Shooting methods solve a BVP by reformulating it as an IVP, using initial values as parameters. The BVP is solved by finding the parameters that reproduces the desired boundary values. For this exercise, we return to considering the BVP of a hanging rope.

Taking as our example the rope BVP (1), how much violence do we have to do to it 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 $x = 0$:

$$\begin{aligned} (1+cx)u'' + cu' &= \rho g, \qquad c = 0.05, \quad \rho g = 0.4 \\ u(0) &= 1 \\ u'(0) &= \alpha \end{aligned} \tag{22}$$

The following exercise attacks this system using a method called "shooting." The strategy behind this method is the following.

- For each value of α, we can solve this problem, and get a numerical solution at a sequence of points up to the right endpoint.
- Since every value of α determines a (numerical) solution $u(5)$, we can regard the difference between the value we got, and the value we want, as a function $F(\alpha) = u(5) - 1.5$. (Where we write $u(5)$, but we should really write something like $u_\alpha(5)$, to emphasize that the solution depends on the parameter.)
- The BVP solution we are looking for has the property that $F(\alpha) = 0$. The Matlab function `fzero` will be used to find α.

As you know, the ODE (1) can be written as a system of first order differential equations. This is done by identifying $y_1 = u$ and $y_2 = u'$, yielding the system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= (\rho g - c y_2)/(1+cx) \end{aligned}$$

For this exercise, you will be using built-in Matlab ODE routines to solve the initial value problem that arises from a guessed initial condition and then you will use the Matlab function `fzero` to solve for the correct initial condition.

Exercise 9:

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

```
function fValue=rope_ode(x,y)
% fValue=rope_ode(x,y) computes the
% rhs of the first-order system

% your name and the date
```

Note that since we plan to use `ode45` there is no need to add the Jacobian matrix to `rope_ode.m`! This is a great convenience, but in many cases you would have to provide a function for the Jacobian or `ode45` (or `ode15s`, *etc.*) might fail. In that case, setting an option allows the Jacobian computation.

(b) Choose a provisional value $\alpha = 0$ and use the Matlab function `ode45` to solve the system (22) on the interval [0,5]. What is the value of $u(5) - 1.5$? Is it positive or negative? (Be careful: which of the components of `y` corresponds with u?)

(c) By trial and error, find a second value of α for which the value of $u(5) - 1.5$ is of the opposite sign as for $\alpha = 0$. The correct value of α lies between the two values you just found.

(d) Write an m-file called `rope_shoot.m` that accepts a value of `alpha` and evaluates `F(alpha)`, for the rope BVP. The file should have the signature

```
function F = rope_shoot ( alpha )
% F = rope_shoot ( alpha )
% comments

% your name and the date
```

and this code should do the following:

- Use the input value of `alpha` as the initial condition for $y'(0)$;
- Use `ode45` to compute the solution `[x,y]` of the IVP (22) defined by the initial conditions, and the right hand side function `rope_ode`, for $x \in [0, 5]$;
- Return in the function value `F` the value of $u(5) - 1.5$.

For a given value of α, the function you just wrote will return $y(5) - 1.5$. When α is just right, it will return 0.

(e) Test that `rope_shoot` returns the same value you obtained above when `alpha=0`.

(f) Use the Matlab function `fzero` to find the value of α that makes $F(\alpha) = y(5) - 1.5 = 0$. `fzero` requires two parameters, a function handle (`@`) first and second the vector `[alpha1,alpha2]` of the two values of α that you just found for which F has opposite signs. What is the value of `alpha` you found?

(g) Plot the solution you found. Does the curve have a height of 1 at $x = 0$ and a height of 1.5 at $x = 5$? You do not need to send me this plot.

(h) Return to your solution for `N=119` in Exercise 2. Use a finite difference expression for the derivative to estimate the derivative of `U2` at the left endpoint. How does it compare with the value α you just computed?

Last change `$Date: 2017/01/16 01:32:00 $`