Lab 4: Boundary Value Problems
MATH2071, University of Pittsburgh, Spring 2023

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 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 (MOL).

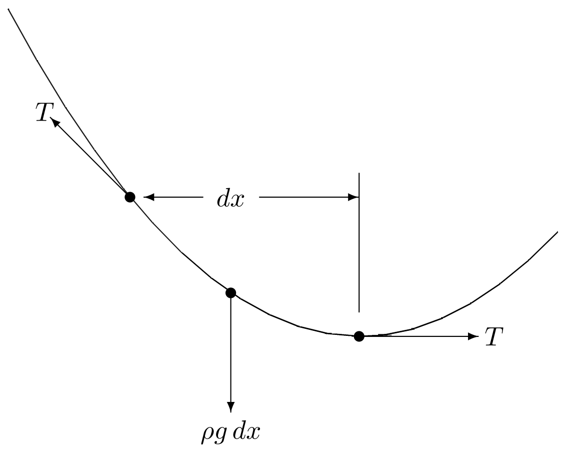
This lab will take four sessions.

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

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

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 $K(x) = 1 + \kappa x$, representing a rope (or spring) whose stiffness varies from end to end. The result is the equation

$$(1+\kappa x)u'' + \kappa u' = \rho g.$$

When $\kappa = 0$, 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 $\kappa = 0.05$, 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

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

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 $K_n = K(x_n) = (1 + \kappa x_n)$.

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

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

and, as above, $\kappa = 0.05$ 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:

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

Actually, in Equation (2), the quantities u_0 and u_5 are not really unknowns, since the boundary conditions give their values explicitly. However, boundary conditions will not always be so simple, and it will be convenient to treat these two quantities as unknown variables, just like the others.

$$\begin{array}{rrrrrrl}
u_0 & 0 & 0 & 0 & 0 & 0 & = 1 \\
(1 + 0.5\kappa\Delta x)u_0 & -2(1 + \kappa\Delta x)u_1 & +(1 + 1.5\kappa\Delta x)u_2 & +0 & +0 & 0 & = 0.4\Delta x^2 \\
0 & (1 + 1.5\kappa\Delta x)u_1 & -2(1 + 2\kappa\Delta x)u_2 & (1 + 2.5\kappa\Delta x)u_3 & +0 & 0 & = 0.4\Delta x^2 \\
0 & 0 & +(1 + 2.5\kappa\Delta x)u_2 & -2(1 + 3\kappa\Delta x)u_3 & +(1 + 3.5\kappa\Delta x)u_4 & 0 & = 0.4\Delta x^2 \\
0 & 0 & +0 & +(1 + 3.5\kappa\Delta x)u_3 & -2(1 + 4\kappa\Delta x)u_4 & (1 + 4.5\kappa\Delta x)u5 & = 0.4\Delta x^2 \\
0 & 0 & 0 & 0 & 0 & u_5 & = 1.5
\end{array}$$

and this system has been formatted to suggest the matrix equation

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ (1+0.5\kappa\Delta x) & -2(1+\kappa\Delta x) & (1+1.5\kappa\Delta x) & 0 & 0 & 0 \\ 0 & (1+1.5\kappa\Delta x) & -2(1+2\kappa\Delta x) & (1+2.5\kappa\Delta x) & 0 & 0 \\ 0 & 0 & (1+2.5\kappa\Delta x) & -2(1+3\kappa\Delta x) & (1+3.5\kappa\Delta x) & 0 \\ 0 & 0 & 0 & (1+3.5\kappa\Delta x) & -2(1+4\kappa\Delta x) & (1+4.5\kappa\Delta x) \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.4\Delta x^2 \\ 0.4\Delta x^2 \\ 0.4\Delta x^2 \\ 0.4\Delta x^2 \\ 1.5 \end{bmatrix} \tag{3}$$

By discretizing the differential equations we have created a set of linear algebraic equations that have the symbolic form $AU = b$.

4 rope_bvp_six_points()

To set up and solve the equations (3), create a file `rope_bvp_six_points.py`, which reads as follows:

```
def rope_bvp_six_points ( uleft = 1.0, uright = 1.5, kappa = 0.05, rhog = 0.4 ):

  import numpy as np

  x1 = 0.0
  x2 = 5.0
  n = 6
  x = np.linspace ( x1, x2, n )
  dx = ( x2 - x1 ) / ( n - 1 )

  A = np.array ( [ \
    [  1,  0,    0,    0,  0,  0 ], \
    [  1+0.5*kappa*dx,  -2*(1+1*kappa*dx), 1+1.5*kappa*dx,  0,  0,  0 ], \
    [  0,  1+1.5*kappa*dx, -2*(1+2*kappa*dx),  1+2.5*kappa*dx,  0,  0 ], \
    [  0,  0,  1+2.5*kappa*dx, -2*(1+3*kappa*dx),  1+3.5*kappa*dx,  0 ], \
    [  0,  0,  0,  1+3.5*kappa*dx, -2*(1+4*kappa*dx),  1+4.5*kappa*dx ], \
    [  0,  0,  0,  0,  0,  1 ] ] )

    b = np.array ( [ \
      uleft , \
      rhog*dx**2, \
      rhog*dx**2, \
      rhog*dx**2, \
      rhog*dx**2, \
      uright ] )

  u = np.linalg.solve ( A, b )

  return x, u
```

5 Exercise 1

1. Create a file `exercise1.py` for the following experiments. Each time you run the code, use the default input values except where indicated, and look at the plot of the results. You only need to turn in the plot for the default values.
2. Run the code with the default values.
3. How does your solution change if you input `kappa = 0.75`?
4. How does your solution change if you input `rhog = 0.0`?
5. How does your solution change if you input `rhog = -0.4`?
6. How does your solution change if you input `uleft=1.0, uright=5.0`?

6 Solving with a general value for N, the number of mesh points

If we want a more accurate and informative solution, we need to increase `n`, the number of mesh points. But we do NOT want to type in the individual entries for an 11×11 or 101×101 matrix! The structure of the matrix is pretty regular, so we can use a `for` loop to do this, if we can express the individual formulas correctly.

We need to fill in an $n \times n$ matrix `A`, and the right hand side `b`. The first (index 0) and last (index `n-1`) equations are special boundary conditions. Equations 1 through `n-2` all have the same format, but are posed at different locations in the mesh. In particular, equation `i` is posed at `x[i]`, and involves only the variables with indices `i-1`, `i`, and `i+1`.

7 rope_bvp.py

Our code will have a rough outline as follows (note that `n` is now an input quantity!):

```
def rope_bvp ( n = 6, uleft = 1.0, uright = 1.5, kappa = 0.05, rhog = 0.4 ):

  Set up x1, x2, x, dx

  Make space for A and b

  A[0,0] = 1
  b[0]   = uleft

  Handle equations 1 through n-2

    A[i,i-1] =
    A[i,i]   =
    A[i,i+1] =
    b[k]     =

  A[n-1,n-1] = 1
  b[n-1]     = uright

  Solve for u

  Return x and u
```

The main challenge here is to figure out the formula for the matrix entries. You should look for a formula that will work in general. We are also deliberately being somewhat vague about the Python details, because it is time for you to get some more confidence in writing Python code.

8 Exercise 2

1. Create a file `exercise2.py` for the following experiments.
2. Run `rope_bvp()` with the default arguments. You should get the exact same results that you got with `rope_bvp_six_points()`.
3. If your results differ, compare the quantities `A` and `b` between the two codes. Something is probably different.
4. If your results for `n=6` look OK, then run the code again with twice the number of intervals, `n=11`. Your result should look smoother.
5. Run your code with `n=121`. It should run quickly, and produce a smooth plot essentially like the previous one.

9 Lagrange piecewise basis functions

For our next topic, our mesh will use an odd number of points `n`, and we will organize that mesh into $m = \frac{n-1}{2}$ subintervals. Each subinterval is bounded by two points with even index, with an odd-index point in the middle. Here is a suggestion of this arrangement, using `n = 11` points, and `m = 5` subintervals:

```
x0  x1  x2  x3  x4  x5  x6  x7  x8  x9  x10
[-------][------][------][------][-------]
```

For our next topic, we will be working with such a mesh of `n` points `x`, and we will want to create a set of special functions $\phi_i(x), \quad 0 \leq i < n$ known as (piecewise quadratic) Lagrange basis functions. Each function ϕ_i is a piecewise quadratic function, and for every mesh point x_j, we have

$$\phi_i(x_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

Note that if the index i is odd (remember, we count starting at 0!), then $\phi_i(x)$ is only nonzero over the single subinterval $[x_{i-1}, x_{i+1}]$. To write it as a formula, we reason as follows:

- $\phi_i(x_{i-1}) = 0$, so $\phi_i(x) = (x - x_{i-1}) \times$ more
- $\phi_i(x_{i+1}) = 0$, so $\phi_i(x) = (x - x_{i-1}) \times (x - x_{i+1}) \times$ more
- $\phi_i(x_i) = 1$, so $\phi_i(x) = \frac{(x-x_{i-1})\times(x-x_{i+1})}{(x_i-x_{i-1})\times(x_i-x_{i+1})}$

If i is even, then $\phi_i(x)$ has the value 1 at a point that is shared by two subintervals, and so we must define it over subinterval $[x_{i-2}, x_i]$ and over $[x_i, x_{i+2}]$, using two quadratic formulas that match at x_i. Using similar reasoning as above, we have

$$\phi_i(x) \quad = \quad \begin{cases} \frac{(x-x_{i-1})(x-x_{i-2})}{(x_i-x_{i-1})(x_i-x_{i-2})} & \text{for } x_{i-2} < x \leq x_i, \\ \frac{(x-x_{i+1})(x-x_{i+2})}{(x_i-x_{i+1})(x_i-x_{i+2})} & \text{for } x_i < x \leq x_{i+2} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

10 lagrange_basis.py

We are going to create a file which allows us to evaluate $\phi_i(x)$. To do this, we need as input the index `i`, the node vector `xn`, and the points at which we want to sample the function, `xs`.

We have to use some tricky python commands here. You don't need to master these commands right now, but you should trust that they are doing what I say:

- `xs = np.atleast_1d ( xs )` allows us to treat `xs` as a vector, even if the user only supplies a scalar. We do this because we will be indexing `xs` later.
- `np.nonzero` allows us to index the entries of a vector where a condition is true (nonzero). In a piecewise formula, we only apply the formula to values of `xs` that are in the particular interval of interest.

Here is a start at the code. You need to finish it by filling in the text where question marks appear.

```python
def phi ( i, xn, xs ):

  import numpy as np

  xs = np.atleast_1d ( xs )
  ns = len ( xs )
  zs = np.zeros ( ns )

  if ( ( i % 2 ) == 1 ):

    ks = np.nonzero ( ( xn[i-1] <= xs ) & ( xs <= xn[i+1] ) )
    zs[ks] =   ( xs[ks] - xn[i-1] ) / ( xn[i] - xn[i-1] ) \
             * ( xs[ks] - xn[i+1] ) / ( xn[i] - xn[i+1] )

  else:

    nn = len ( xn )

    if ( 0 < i ):
      ks = np.nonzero ( ( xn[i-2] <= xs ) & ( xs <= xn[i] ) )
      zs[ks] = ( xs[ks] - xn[i-2] ) / ( xn[i] - xn[i-2] ) \
             * ( xs[ks] - xn[i-1] ) / ( xn[i] - xn[i-1] )

    if ( i < nn - 1 ):
      ks = ???
      zs[ks] = ???

  return zs
```

Why do we have the checks `0 < i` and `i < nn - 1` for the case when `i` is even?

11 Exercise 3

- Create a file `exercise3.py` for the following experiments.
- Consider the interval $[2, 8]$. Define a mesh of $n = 7$ points over this interval.
- Set up an array `xs` of 101 points in the interval.
- Evaluate $ys = \phi_0(xs)$ at each of these points, and plot `(xs,ys)`.
- Now evaluate $ys = \phi_1(xs)$ in the same way.
- Now evaluate $ys = \phi_2(xs)$. This is also a basis function of even index, but unlike $\phi_0(xs)$ it should now extend over a wider range.

At each step, check that your plot seems to correspond to our definitions of even and odd piecewise quadratic Lagrange basis functions. If not, your code may have an error and you need to find it and fix it before proceeding.

12 Lagrange basis function derivatives

It turns out that we will also need to compute the derivatives of the piecewise quadratic Lagrange basis functions. To do this, we will add a second definition, called `phip()`, to `lagrange_basis.py`. It will look very similar to `phi()`, and so the easiest thing to do is to start with a copy of that function and modify it. All you need to do is write the formulas for the derivatives. Here is a start:

```
def phip ( i, xn, xs ):

  import numpy as np

  xs = np.atleast_1d ( xs )
  ns = len ( xs )
  zps = np.zeros ( ns )

  if ( ( i % 2 ) == 1 ):

    ks = np.nonzero ( ( xn[i-1] <= xs ) & ( xs <= xn[i+1] ) )
    zps[ks] =     1.0                 / ( xn[i] - xn[i-1] ) * ( xs[ks] - xn[i+1] ) / ( xn[i]
        - xn[i+1] ) \
            + ( xs[ks] - xn[i-1] ) / ( xn[i] - xn[i-1] ) * 1.0             / ( xn[i]
                - xn[i+1] )

  else:

    nn = len ( xn )

    if ( 0 < i ):
      ks = ???
      zps[ks] = ???

    if ( i < nn - 1 ):
      ks = ???
      zps[ks] = ???

  return zps
```

Finish the definition of `phip()`.

13 Exercise 4

In order to have confidence in your `phip()` definition, we will now plot a couple of these functions. Before you see the plot, you might ask yourself what you expect to see.

- Create a file `exercise4.py` for the following experiments.
- Consider the interval $[2, 8]$. Define a mesh of $n = 7$ points over this interval.
- Set up an array `xs` of 101 points in the interval.
- Evaluate $yps = \phi'_0(xs)$ at each of these points, and plot `(xs,yps)`.
- Now evaluate $yps = \phi'_1(xs)$ in the same way.
- Now evaluate $yps = \phi'_2(xs)$. This is also a basis function of even index, but unlike $\phi'_0(xs)$ it should now extend over a wider range.

You may find something slightly surprising about these plots. They may look almost discontinuous. In fact, the actual functions are discontinuous. Can you explain this?

14 fem_value()

Suppose that we use our piecewise quadratic Lagrange basis functions as building blocks, and we construct a new function $y(x) = \sum_{i=0}^{i<n} cn_i \phi_i(x)$. If we have the values of the coefficients `cn`, the nodes `xn`, and a list of evaluation points `xs`, we can easily add up a linear combination of the $\phi_i()$ basis functions.

Write a code `fem_value.py` which contains the following definition:

```
def fem_value ( cn, xn, xs ):

  from lagrange_basis import phi
  import numpy as np

  nn = len ( xn )

  ns = len ( xs )
  ys = np.zeros ( ns )

  for i in range ( 0, nn ):
    ys[j] = ys[j] + ???

  return ys
```

after you replace the question marks by the appropriate formula for the linear combination.

15 Exercise 5

- Create file `exercise5.py` for this experiment.
- Assume we are interested in the interval `[2,8]`, with `nn=7` equally spaced mesh points, and a coefficient vector `cn = [ 1.0, 0.68695865, 0.42029673, 0.23631118, 0.23352226, 0.54437828, 1.5 ]`. Use your `fem_value()` function to evaluate the finite element function `ys` whose coefficients are `cn`, at `ns=101` equally spaced points in the interval. Plot your result. It should look familiar.
- Modify your file `fem_value.py` by adding a second definition,

```
def femp_value ( cn, xn, xs ):
```

  which evaluates the *derivative* $\frac{dy}{dx}$ of the finite element function `ys`. Evaluate this function at 101 points over the interval and plot it. This should *not* look familiar, and parts of the plot might seem slightly surprising at first. You should be able to explain what is happening.

Note that `ys(2.0)=1.0`, `ys(3.0) = 0.68695865`, and so on. Explain why this is **not** a coincidence!

16 Approximate integration

The final tool we will need in order to carry out our estimated solution of a boundary value problem allows us to estimate integrals. In Math2070 we spent some time looking at methods for approximating $q = \int f(x)dx$, the integral of a function over some interval. A quadrature rule supplies a set of]ttnq points `xq`, and weights `wq`, so that $q \approx \sum_{i=0}^{i<nq} wq_i f(xq_i)$.

A composite rule usually does a better job by using a set of `nn` mesh points `xn` to break the interval up into subintervals. The `j`-th subinterval is `[x[j],x[j+1]]`, but we'll think of it as `[a,b]`. Then we have to recompute the points and weights for this interval as xab, wab, estimate $qab \approx \sum_{i=0}^{i<nq} wab_i f(xab_i)$. and add each of these estimates to get our final integral estimate `q`.

The points and weights for a Gauss quadrature using `nq` points are available as follows:

```
  from numpy.polynomial.legendre import leggauss
  xq, wq = leggauss ( nq )
```

We will now create a function that uses a composite Gauss quadrature rule for a function over an interval for which we have supplied a mesh.

17 gauss_quad()

Create the following file `gauss_quad.py`, which will allow us to approximate the integral of a function `func(x)` over an interval that has been meshed with mesh points `xn`, using a Gauss quadrature rule involving `nq` points.

```
def gauss_quad ( func, xn, nq ):

  from numpy.polynomial.legendre import leggauss
  import numpy as np

  xq, wq = leggauss ( nq )

  q = 0.0
  nn = len ( xn )

  for i in range ( 0, nn - 1 ):
    xab = ( xn[i] * ( 1.0 - xq ) + xn[i+1] * ( xq + 1.0 ) ) / 2.0
    wab = ( xn[i+1] - xn[i] ) * wq / 2.0
    fab = func ( xab )
    qab = np.sum ( wab * fab )
    q = q + qab
  return q
```

The values `xq, wq` are integration points and weights for the interval [-1,+1]. The integral estimate is made by doing a separate estimate over each of the `nn-1` intervals. Inside interval `i`, the values `xab, wab` are computed for the interval `[x[i],x[i+1]]`. The function is evaluated at the points `xab`, and then added to the integral estimate `q` using the weights `wab`.

18 Exercise 6

- Create the file `exercise6.py` for this experiment.
- Insert `def humps(x):` so that $y(x) = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6$
- Insert `def humps_anti(x):` so that $ya = 10\arctan(10(x-0.3)) + 5\arctan(5(x-0.9)) - 6x$
- Set `exact = humps_anti(2.0) - humps_anti(0.0)`;
- Create a double for loop on `nn` and `nq` as follows:
- for `nn = 2, 4, 8, 16, 32`, define a set `xn` of equally spaced points over [0,2]
- for `nq = 1, 2, 4, 8, 16`, use `gauss_quad()` to estimate $q = \int_0^2 humps(x)dx$.
- Print each pair of values `nn, nq`, and the integration error, $||q - exact||$;

In some sense, the "cost" of the integral approximation is $nn * nq$. There are 5 integral estimates with a cost of 32 function evaluations. Which one does the best?

19 Integral of a finite element function

As part of the finite element method, it is necesary to compute integrals involving the product of a finite element function $f(x) = \sum_{j=0}^{j=n-1} c_j \phi_j(x)$ or its derivative $f'(x)$, multiplied by a basis function $\phi_i(x)$ or its derivative $\phi_i'(x)$. Before we consider a complete finite element program, then, it is worth while to see the details of how such an integral is computed, using the tools we have developed so far.

In particular, for the example of the rope boundary value problem, we need to set up a linear system $A * c = b$, where `c` is the vector of finite element coefficients, `A` can be computed in two parts, `A = A1 + A2`, and `b` is a right hand side vector. Most of the linear system components can be computed once we choose the indices `i` and `j`:

$$
\begin{aligned}
A1_{i,j} &= \int_2^8 -(1+\kappa x)\,\phi_j'(x)\,\phi_i'(x)\,dx \\
A2_{i,j} &= \int_2^8 \kappa\,\phi_j'(x)\,\phi_i(x)\,dx \\
b_i &= \int_2^8 \rho\,g\,\phi_i(x)\,dx
\end{aligned}
$$

As before, the first and last rows of the linear system will be determined instead by boundary conditions.

We are going to use `gauss_quad()` to approximate these integrals. This process was simple when the integrand was a simple formula involving one variable, as when we were working with `y=humps(x)` so that we could just write

```
q = gauss_quad ( humps, xn, nq )
```

but now our integrands are more complicated. They involve a Python expression of several variables, in which the integration must be identified. To compute `A1`, we might want to write:

```
q = gauss_quad ( -(1+ kappa * x) * phip(j,xn,x) * phip(i,xn,x), xn, nq )
```

but the integrand seems to involve variables `kappa, x, j, xn,` and `i`. We need to tell `gauss_quad` which variable is the free variable or integration variable. We do that with another *lambda expression*:

```
q = gauss_quad ( lambda x: -(1+ kappa * x) * phip(j,xn,x) * phip(i,xn,x), xn, nq )
```

and now `gauss_quad()` knows to integrate over x, leaving the other variables at their input values.

20 Exercise 7

- Create a file `exercise7.py` for this experiment.
- As before, assume that we are working on the interval [2,8], and have created meshpoints `xn` using `nn = 7` points.
- As before, assume `kappa = 0.05`, `rhog = 0.4`;
- Assume we have a finite element function whose coefficients are `cn = [ 1.0, 0.68695865, 0.42029673, 0.23631118, 0.23352226, 0.54437828, 1.5 ]`.
- For the following steps, use `gauss_quad()` with `nq = 3` to estimate integrals;
- Estimate the integral $A1_{2,3}$, correct result is 1.766....
- Estimate the integral $A2_{2,3}$, correct result is 0.033...
- Estimate the integral b_2, correct result is 0.222...

21 The Finite element method

Now let's apply the finite element method to our boundary value problem. We are working on the interval [2,8], and recall that our system is:

$$\begin{aligned}(1+\kappa x)u'' + \kappa u' &= \rho g \\ \kappa &= 0.05 \\ \rho g &= 0.4 \\ u(2.0) &= 1 \\ u(8.0) &= 1.5\end{aligned} \tag{5}$$

We will shortly be using integration, for which it is crucial that the state equation can be rewritten in what is sometimes called "self-adjoint form":

$$\frac{d}{dx}\left((1+\kappa x)u'\right) = \rho g$$

We have defined `xn`, a mesh of `nn` points, and our basis functions $\phi_i(x)$ for $0 \le i < nn$. We want to compute an approximate solution $y(x) = \sum_{j=0}^{j<nn} c_j \phi_j(x)$. We plan to write a linear system $A * c = b$ that will give us the coefficient values `c`. The first and last equations in this linear system specify the solution value at the endpoints:

$$c_0 = 1$$
$$c_{nn-1} = 1.5$$

The equations for $2 \le i \le nn-2$ will all be formed in a common way, so let's assume we are interested in some equation i and see how it is set up.

1. Rewrite the original system by replacing $y(x)$ by its finite element sum:

$$(1+\kappa x)\sum_{j=0}^{j<nn} c_j \phi"_j(x) + \kappa \sum_{j=0}^{j<nn} c_j \phi'_j(x) = \rho g$$

2. Premultiply the equation by $\phi_i(x)$:

$$\phi_i(x)\left((1+\kappa x)\sum_{j=0}^{j<nn} c_j \phi"_j(x) + \kappa \sum_{j=0}^{j<nn} c_j \phi'_j(x)\right) = \phi_i(x)\rho g$$

3. Integrate the equation over the interval:

$$\int_2^8 \phi_i(x)\left((1+\kappa x)\sum_{j=0}^{j<nn} c_j \phi"_j(x) + \kappa \sum_{j=0}^{j<nn} c_j \phi'_j(x)\right) dx = \int_2^8 \phi_i(x)\rho g\, dx$$

4. Rewrite the equation in its "self-adjoint" form:

$$\int_2^8 \phi_i(x)\left(\frac{d}{dx}(1+\kappa x)\sum_{j=0}^{j<nn} c_j \phi'_j(x)\right) dx = \int_2^8 \phi_i(x)\rho g\, dx$$

5. Use integration by parts so we don't have to compute $\phi"(x)$. Note that boundary term is zero.

$$\int_2^8 \left(-\phi'_i(x)(1+\kappa x)\sum_{j=0}^{j<nn} c_j \phi'_j(x)\right) dx = \int_2^8 \phi_i(x)\rho g\, dx$$

6. Pull out summation to left, and c_j to right:

$$\sum_{j=0}^{j<nn}\left(\int_2^8 -\phi'_i(x)(1+\kappa x)\phi'_j(x)\, dx\right) c_j = \int_2^8 \phi_i(x)\rho g\, dx$$

7. Realize that we have the i-th equation of a linear system $A * c = b$:

$$\sum_{j=0}^{j<nn} A_{i,j} c_j = b_i$$

8. Use Gauss quadrature to evaluate the integrals $A_{i,0}, A_{i,1}, ..., A_{i,nn-1}, b_i$
9. Solve the linear system.

22 Finite element program for the rope

Here is the sketch of a program to apply the finite element method to our rope BVP. We have allowed the number of mesh points `nn` to be an input argument, with a default value of 7.

```
def exercise8 ( nn = 7 ):

#  Import stuff

#  Define the problem data

#  Det the mesh
#  Get a Gauss rule

#  Create the linear system

  A = np.zeros ( [ nn, nn ] )
  b = np.zeros ( nn )

  for i in range ( 0, nn ):

    if ( i == 0 ):
      A[i,i] = 1.0
      b[i] = uleft
    elif ( i == nn - 1 ):
      A[i,i] = 1.0
      b[i] = uright
    else:
      b[i] = gauss_quad ( lambda xs: phi(i,xn,xs) * rhog, xn, nq )
      for j in range ( 0, nn ):
        A[i,j] = A[i,j] + gauss_quad ( lambda xs: \
          phip(i,xn,xs) * (-1.0-kappa*xs) * phip(j,xn,xs)

#  Solve A*c=b

  c = ?

#  Compute the solution curve (xs,ys) and plot it:

  xs =
  ys =

  return
```

23 Exercise 8

- Create a file `exercise8.py` by copying the text above, and filling in all the gaps.
- Run your program using the usual mesh of `nn=7`. You should expect to see a plot very similar to what we found in exercises 1 and 5.
- The plot in exercise 1 seems crude, but your new plot should look smooth, but both plots are based on a very crude mesh of only 6 or 7 points. What explains the difference?

Congratulations! You've solve a boundary value problem using the finite element method.

24 A new boundary value problem

Consider the following boundary value problem:

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

defined over the interval [0,1], with boundary values

```
y[0.0] = yleft = 0.0
y[1.0] = yright = 0.0
```

which has the exact solution

$$y_{exact}(x) = x - e^{(1-x)/2} \sin(\omega x) / \sin(\omega)$$

with $\omega = \sqrt{3}/2$.

We will try to set up a finite element approach for solving this problem. It will be very similar to what we did for the previous exercise, but now you have to set the data for this new problem, and make some modifications to the definition of the linear system.

25 Exercise 9

- Create a file `exercise9.py` for this experiment with signature

```
def exercise9 ( nn = 9 ):
```

- Start from a copy of `exercise8.py`, and try to modify it so that it solves the new boundary value problem.
- Your computation of the A_ij matrix entries will now be the sum of three terms rather than two, involving products $\phi_i'(x)\phi_j'(x)$, $\phi_i(x)\phi_j'(x)$ and $\phi_i(x)\phi_j(x)$;
- Remember that the first term is defined after integration by parts is used;
- Use `nn=9` for your grid.
- Make a plot with `ns=26` sample values, showing both the finite element solution (red dots, 'ro') and the exact solution (blue line, 'b-').
- For the same 26 sample values, compute and print `e`, the maximum absolute value of the difference between your finite element solution and the exact solution.

26 Exercise 10

- Create a file `exercise10.py` for this experiment, starting from a copy of `exercise9.py`, with signature

```
def exercise10 ( nn = 9 ):
```

- Turn off the plotting feature.
- Change the code so that it returns the value of `e`, the maximum absolute value of the difference between your finite element solution and the exact solution at the 26 evenly spaced points.
- Define an array `nn` $=$ 3, 5, 9, 17, 33, and a array `e` of size 5 that is initially zero.
- For $0 \leq k < 5$, compute and print `e[k] = exercise10(nn[k]`;
- For $0 \leq k < 4$, compute and print `r[k] = e[k] / e[k+1]`

On each step, we are doubling the number of intervals, and thus dividing the interval size h by 2. The behavior of `r[k]` tells us the corresponding divisor for the error. For a linear convergence rate, we would expect to see `r[k]` approximately 2. What do we see instead, and how do you interpret it?