CE 594 - Final Project

This project and all associated code are available on Tristan's website:

http://atdyer.github.io/ncsu/courses/ce594/project/index.html

For this project, we are going to analyze a seepage problem. Imagine there is a sand filled pipe buried underground below the water table and oriented from left to right. The pipe has a cross-sectional area of 2m^2 and is 10m long. Water is flowing into the left end of the pipe at a rate of 100m^3/day. At the right end of the pipe we measure the hydraulic head, h, (look it up on wiki if you need to) to be 3m. The wall of the pipe is made of a permeable material so that water can pass through, which effectively creates a source or sink depending on the hydraulic head inside the pipe. We will represent this as a source term of the form $\alpha^0 h$.

The governing differential equation can be derived from Darcy's Law, which relates the volume rate of flow per unit area to the rate of change of the hydraulic head, given by

$$q = -k\frac{dh}{dx}$$

where k is the hydraulic conductivity. For a steady state flow and assuming that water is incompressible, conservation of mass tells us that the amount of water flowing into any part of the pipe must be equal to the amount of water flowing out. In other words, any change in the volume rate of flow must balance any water gained or lost through a source or sink, respectively. Mathematically, this is stated as

$$\frac{d}{dx}(Aq) - s = 0$$

where s is the source term. For this problem we can replace$s$ with $\alpha^0 h$.

Using the following parameters

1. State the governing strong form problem, identifying essential and natural boundary conditions
2. Derive the weak form
3. Define appropriate approximation spaces and the Galerkin problem
4. Develop the matrix equations showing how all coefficients are calculated
5. Modify your finite element code to solve this problem using at least 8 elements
6. Plot your solution and state the hydraulic head at the left end of the pipe and the amount of water flowing out of the right end

$$\begin{aligned} A &= 2\text{m}^2 \\ k &= 250\text{m/day} \\ \alpha^0 &= -5\text{m/day} \end{aligned}$$

Solution

The complete hand-written solution can be found in the following pdf document:

Solution

The hand-written solution contains the complete derivation of the matrix form starting from the strong form. Below, the strong form, weak form, Galerkin form, and matrix form are shown for reference.

Strong Form

$$\frac{d}{dx}(-Ak\frac{du}{dx}) - S = 0$$

$$u(L) = 3$$

$$\frac{du}{dx}(0) = -0.2$$

Weak Form

$$\int_{\Omega} \frac{dw}{dx} Ak \frac{du}{dx} dx = \int_{\Omega} wS dx - wAk \frac{du}{dx}\bigg|_{\Gamma^h}$$

Galerkin Form

$$a(w^h, v^h) - (w^h, v^h) = (w^h, g^h) - a(w^h, g^h) - w^h Ak \frac{du}{dx}\bigg|_{\Gamma^h}$$

Matrix Form

$$([K] - [M])\{d\} = [F]$$

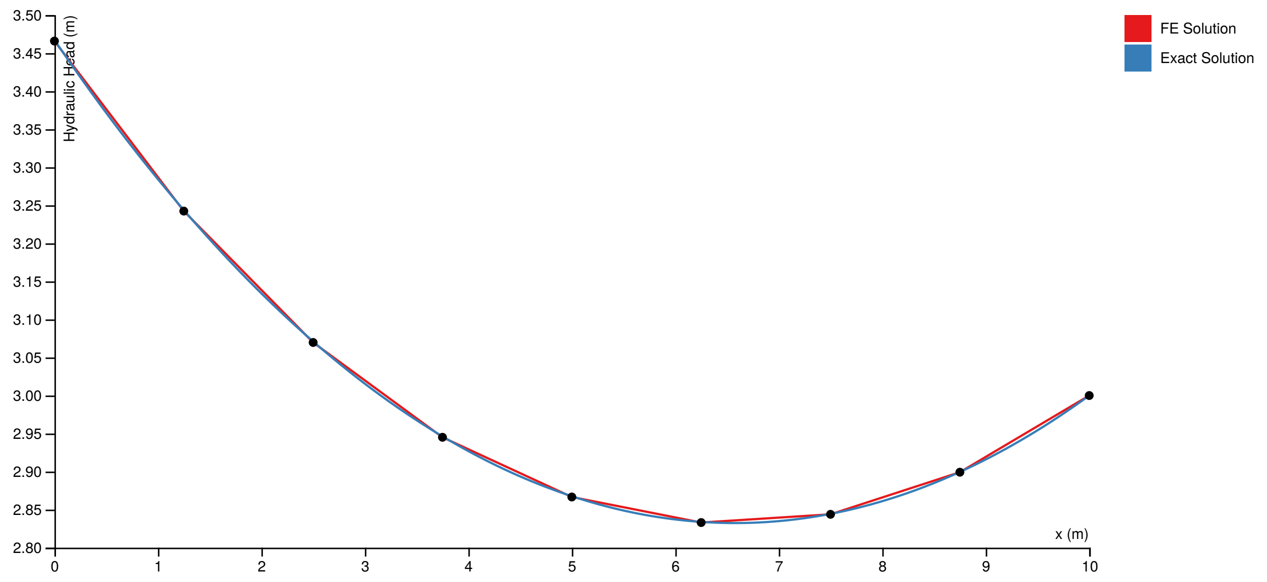
Code and Implementation

The code used to solve the matrix form of the equation is written in Python and composed of 3 scripts. A fourth script, containing the problem definition is used to execute the code and plot the results.

- **project.py** - This script contains the problem definition. Run this script to solve the problem and generate a plot.
- **fe.py** - This is the finite element implementation used to solve this problem. It builds the matrices defined in the above matrix form of the governing equation and solves for the profile of the hydraulic head along the length of the pipe.
- **jacobian.py** - This script contains functions used in coordinate transformations.
- **shape_functions.py** - This script contains a function that can be used to generate shape functions and their derivatives for any number of element nodes. Further implementation details can be found on the page.

Results

The following interactive plot shows the results of the finite element code compared to the exact solution. Use the number pickers below the plot to change the number of element nodes and number of elements.



Number of Element Nodes: 2 **Number of Elements:** 8

We can see that as the number of element nodes and number of elements increase, the finite element solution converges on the exact solution, and we're able to calculate the following values:

- Hydraulic head at the left end of the pipe:**3.47m**
- Flow rate at the right end of the pipe:**-49.43m^3/day**

Let's say hydraulic head is $u(x)$

STRONG FORM

$$\frac{d}{dx}\left(-Ak\frac{du}{dx}\right) - s = 0$$

(S)

$$u(L) = 3 \quad \leftarrow \text{essential b.c.}$$

$$\frac{du}{dx}(0) = -0.2 \quad \leftarrow \text{natural b.c.}$$

Define :

$$S = \{ u \mid u \in H^1,\ u(L) = 3 \}$$
$$V = \{ w \mid w \in H^1,\ w(L) = 0 \}$$

- Multiply both sides by -1 to get strong form as:

$$\frac{d}{dx}\left(Ak\frac{du}{dx}\right) + s = 0$$

- Multiply both sides by the weight function and integrate:

$$\int_\Omega w\left[\frac{d}{dx}\left(Ak\frac{du}{dx}\right)\right] dx + \int_\Omega ws\,dx = 0$$

- Integration by parts on first term :

$$\left(\begin{array}{l} \int_a^b f g'\,dx = fg\Big|_a^b - \int_a^b f' g\,dx \\ \quad f = w \\ \quad g = Ak\,du/dx \end{array}\right.$$

$$wAk\frac{du}{dx}\Big|_0^L - \int_\Omega \frac{dw}{dx}Ak\frac{du}{dx}\,dx + \int_\Omega ws\,dx = 0$$

- Rearranging :

$$\int_\Omega \frac{dw}{dx} Ak \frac{du}{dx} dx = wAk\frac{du}{dx}\Big|_0^L + \int_\Omega ws\, dx$$

- Since $w(L) = 0$,

$$\int_\Omega \frac{dw}{dx} Ak \frac{du}{dx}\, dx = -wAk\frac{du}{dx}\Big|_{x=0} + \int_\Omega ws\, dx$$

<u>WEAK FORM</u>

(W) Find $u \in S$ such that $\forall\ w \in V$:

$$\int_\Omega \frac{dw}{dx} Ak \frac{du}{dx}\, dx = \int_\Omega ws\, dx - wAk\frac{du}{dx}\Big|_{x=0}$$

Define:

$S^h \subset S$, i.e. if $u^h \in S^h$, then $u^h \in S$

$V^h \subset V$, i.e. if $v^h \in V^h$, then $v^h \in V$

where the h superscript refers to a discretization of the domain Ω, which is parameterized by a characteristic length scale, h.

Thus we can infer that:

$$u^h = g \text{ on } \Gamma_g$$
$$w^h = 0 \text{ on } \Gamma_g$$

Given a function $v^h \in V^h$ and a function g^h which satisfies the natural boundary conditions, we can define u^h to be :

$$u^h = v^h + g^h$$

- Note that u^h still satisfies the conditions of S that $u = g$ on Γ_g :

$$u^h\big|_{\Gamma_g} = v^h\big|_{\Gamma_g} + g^h\big|_{\Gamma_g}$$

$$= 0 + g = g \checkmark$$

- Substitute the definitions of u^h and w^h into the weak form:

$$\int_\Omega \frac{dw^h}{dx} Ak \frac{d(v^h + g^h)}{dx}\, dx = \int_\Omega w^h s\, dx - w^h Ak \frac{du}{dx}\bigg|_{x=0}$$

- The source term for this problem includes u, so substitute u in, and perform u^h substitution:

$$\int_\Omega \frac{dw^h}{dx} Ak \frac{d(v^h+g^h)}{dx}\, dx = \int_\Omega w^h \alpha_0 h\, dx - w^h Ak \frac{du}{dx}\bigg|_{x=0}$$

$$\int_\Omega \frac{dw^h}{dx} Ak \frac{d(v^h+g^h)}{dx}\, dx = \int_\Omega w^h \alpha_0 (v^h + g^h)\, dx - w^h Ak \frac{du}{dx}\bigg|_{x=0}$$

- Expanding ...

$$\int_\Omega \frac{dw^h}{dx} Ak \frac{dv^h}{dx} dx + \int_\Omega \frac{dw^h}{dx} Ak \frac{dg^h}{dx} dx = \int_\Omega w^h \alpha_0 v^h dx + \int_\Omega w^h \alpha_0 g^h dx - w^h Ak \frac{du}{dx}\bigg|_{x=0}$$

- Rearranging ...

$$\int_\Omega \frac{dw^h}{dx} Ak \frac{dv^h}{dx} dx - \int_\Omega w^h \alpha_0 v^h dx = \int_\Omega w^h \alpha_0 g^h dx - \int_\Omega \frac{dw^h}{dx} Ak \frac{dg^h}{dx} dx - w^h Ak \frac{du}{dx}\bigg|_{x=0}$$

- DEFINE :

$$a(f,g) = \int_\Omega \frac{df}{dx} Ak \frac{dg}{dx} dx \qquad (f,g) = \int_\Omega f \alpha_0 g\, dx$$

Using these definitions, we can write the Galerkin form as:

$$(G) \qquad a(w^h, v^h) - (w^h, v^h) = (w^h, g^h) - a(w^h, g^h) - w^h Ak \frac{du}{dx}\Big|_{x=0}$$

To get to the matrix form, let's first define some notations:

η = set of all nodes
η_g = nodes at boundary conditions (essential)
$\eta_f = \eta \backslash \eta_g$ = nodes at dofs

If $w^h \in V^h$, then there exists constants, C_A, for $A = 1, 2, \dots, n$ such that

$$w^h = \sum_{A=1}^{n} C_A N_A$$

where N_A must satisfy

$$N_A\Big|_{\eta_g} = 0 \qquad \text{(ie } = 0 \text{ at essential b.c.s)}$$

an C_A is an arbitrary constant. This can therefore be written as a sum of the non-boundary nodes:

$$w^h = \sum_{A \in \eta_f} C_A N_A$$

The same can be applied to v^h and g^h, keeping in mind that $v^h + g^h$ must equal u^h, giving us the following:

$$v^h = \underset{\substack{\uparrow \\ \text{Note this is at dofs}}}{\sum_{B=\eta_f}} d_B N_B \qquad g^h = \underset{\substack{\uparrow \\ \text{Note this is at essential b.c.}}}{\sum_{B=\eta_g}} d_B N_B$$

Now substitute into the Galerkin form:

$$a\left(\sum_{A\in\eta_f} c_A N_A, \sum_{B\in\eta_f} d_B N_B\right) - \left(\sum_{A\in\eta_f} c_A N_A, \sum_{B\in\eta_f} d_B N_B\right)$$

$$=$$

$$\left(\sum_{A\in\eta_f} c_A N_A, \sum_{B\in\eta_g} d_B N_B\right) - a\left(\sum_{A\in\eta_f} c_A N_A, \sum_{B\in\eta_g} d_B N_B\right) - \underset{\uparrow}{Ak\frac{du}{dh}}\left[\sum_{A\in\eta_f} c_A N_A\right]_{\Gamma_h}$$

Recall that this is a known value at Γ_h

Because c_A is arbitrary, we can choose $c_A = 1$ for a single value of A and $c_A = 0$ for all others, resulting in:

$$c_A\, a\left(N_A, \sum_{B\in\eta_f} d_B N_B\right) - c_A\left(N_A, \sum_{B\in\eta_f} d_B N_B\right)$$

$$=$$

$$c_A\left(N_A, \sum_{B\in\eta_g} d_B N_B\right) - c_A\, a\left(N_A, \sum_{B\in\eta_g} d_B N_B\right) - c_A N_A Ak\frac{du}{dh}\Big|_{\Gamma_h}$$

Divide the entire equation by c_A to get:

$$a\left(N_A, \sum_{B\in\eta_f} d_B N_B\right) - \left(N_A, \sum_{B\in\eta_f} d_B N_B\right)$$

$$=$$

$$\left(N_A, \sum_{B\in\eta_g} d_B N_B\right) - a\left(N_A, \sum_{B\in\eta_g} d_B N_B\right) - N_A Ak\frac{du}{dh}\Big|_{\Gamma_h}$$

Using the bilinearity of $(\cdot,\cdot)$ and $a(\circ,\circ)$, this becomes :

$$\sum_{B\in\eta_f} a(N_A,N_B)\,d_B - \sum_{B\in\eta_f} (N_A,N_B)\,d_B$$

$$=$$

$$\sum_{B\in\eta_g} (N_A,N_B)\,d_B - \sum_{B\in\eta_g} a(N_A,N_B)\,d_B - N_A A k \frac{du}{dh}\Big|_{\Gamma_h}$$

Now define the following :

$$K_{AB} = a(N_A,N_B)$$

$$M_{AB} = (N_A,N_B)$$

$$F = \sum_{B\in\eta_g} (N_A,N_B)d_B - \sum_{B\in\eta_g} a(N_A,N_B)\,d_B - N_A A k \frac{du}{dh}\Big|_{\Gamma_h}$$

And we're left with :

$$\sum_{B\in\eta_f} K_{AB}\,d_B - \sum_{B\in\eta_f} M_{AB}\,d_B = F$$

Which can be simplified and written in matrix form as:

(M) $$\left([K]-[M]\right)\{d\} = [F]$$

The complete code, shown below, can also be found on **Github**

```
import matplotlib.pyplot as plt
from fe import solve_fe
from math import exp

### Constants
A = 2.0
k = 250.0
l = 10.0
alpha = -5.0

### Mesh
num_elements = 1
num_element_nodes = 8
num_nodes = num_elements * ( num_element_nodes - 1 ) + 1

### Boundary conditions
# Essential
bc_essential = dict()
bc_essential[ num_nodes - 1 ] = 3.0

# Natural
bc_natural = dict()
bc_natural[ 0 ] = -0.2

### Solve
d, x = solve_fe( A, k, l, alpha, num_elements, num_element_nodes, bc_essential, bc_natu

### Exact solution as a function
def exact ( _xe ):
    return exp( -_xe / 10.0 ) * ( 2.73368 + 0.733676 * exp( _xe / 5.0 ) )

xe = [ float(i) * ( float(l) / ( 250 - 1 ) ) for i in range( 250 ) ]
ye = [ exact( _x ) for _x in xe ]


### Plot
plt.plot( x, d, label='FE Solution' )
plt.plot( xe, ye, label='Exact Solution')
plt.legend()
plt.show()
```

The complete code, shown below, can also be found on **Github**

```
import numpy as np
from scipy.integrate import quadrature as integrate
from shape_functions import shape_functions
from jacobian import jacobian

#
# A: area
# k: hydraulic conductivity
# l: length
# alpha: alpha term associated with pipe permeability
# num_elements: number of elements
# num_element_nodes: number of nodes per element
# bc_essential: ( dict ) essential boundary conditions
# bc_natural: ( dict ) natural boundary conditions
#
def solve_fe ( A, k, l, alpha, num_elements, num_element_nodes, bc_essential, bc_natura

    # Calculate number of nodes
    num_nodes = num_elements * ( num_element_nodes - 1 ) + 1

    # Get x-coordinate for each node
    x_coord = [ float(i) * ( float(l) / float( num_nodes - 1 ) ) for i in range( num_nc

    # Get the shape functions
    N, dN, xi = shape_functions( num_element_nodes )

    # Create IEN and ID functions
    def IEN ( element_number, local_node_number ):
        return ( num_element_nodes - 1 ) * element_number + local_node_number

    # Create global arrays
    K = np.zeros( ( num_nodes, num_nodes ) )
    M = np.zeros( ( num_nodes, num_nodes ) )
    F = np.zeros( ( num_nodes, 1 ) )

    # Element loop
    for element in range( num_elements ):

        # Create local matrices
        ke = np.zeros( ( num_element_nodes, num_element_nodes ) )
        me = np.zeros( ( num_element_nodes, num_element_nodes ) )
        fe = np.zeros( ( num_element_nodes, 1 ) )

        # Get globals
        nodes = [ IEN( element, i ) for i in range( num_element_nodes ) ]
        x = [ x_coord[ node ] for node in nodes ]

        # Coordinate transformation and jacobian for this element
        J = jacobian( x[ 0 ], x[ len( x ) - 1 ] )

        # Node loop
        for row in range( num_element_nodes ):

            # Check for natural boundary condition contribution
            if nodes[ row ] in bc_natural:

                fe[ row, 0 ] -= N[ row ]( xi[ row ] ) * bc_natural[ nodes[ row ] ] * A

            # Other node noop
            for col in range( num_element_nodes ):

                # Integrate to calculate ke term
                _k = integrate( lambda _xi: ( dN[row](_xi) / J )*( dN[col](_xi) / J ),
                                xi[0],
                                xi[len(xi)-1] )[ 0 ]

                _m = integrate( lambda _xi: N[ row ]( _xi ) * N[ col ]( _xi ),
                                xi[0],
                                xi[len(xi)-1] )[ 0 ]

                # Add the term to ke, converting to local coordinates
                ke[ row, col ] = _k * A * k * J
                me[ row, col ] = _m * alpha * J

        # Update global arrays with contributions from local
        for row in range( num_element_nodes ):

            # Get the global row
            r = IEN( element, row )

            # Add local f to global F
            F[ r, 0 ] += fe[ row, 0 ]

            for col in range( num_element_nodes ):

                # Get the global column
                c = IEN( element, col )

                # Add the local k to global K
                K[ r, c ] += ke[ row, col ]
                M[ r, c ] += me[ row, col ]

    # Apply essential boundary conditions using elimination method
    K_mask = np.ones( K.shape, dtype=bool )
    F_mask = np.ones( F.shape, dtype=bool )
    dof = num_nodes - len( bc_essential )
    for node, value in bc_essential.iteritems():

        # Mask row and column in the K matrix and M matrix
        K_mask[ node, : ] = False
        K_mask[ :, node ] = False

        # Mask row in the F array
        F_mask[ node, : ] = False

        # Subtract from the F array
        for row in range( num_nodes ):

            _k = K[ row, node ]
            _m = M[ row, node ]
            F[ row, 0 ] -= _k * value
            F[ row, 0 ] += _m * value

    # Apply the mask and reshape
    K = np.reshape( K[ K_mask ], ( dof, dof ) )
    M = np.reshape( M[ K_mask ], ( dof, dof ) )
    F = np.reshape( F[ F_mask ], ( dof, 1 ) )
```

```python
    # Solve for unknowns
    d = np.linalg.solve( K-M, F )

    # Place essential boundary condition values into solution
    for node, value in bc_essential.iteritems():

        d = np.insert( d, node, value )

    return d, x_coord
```

The complete code, shown below, can also be found on **Github**

```
# Tranform 1d coordinate from normalized value on [-1,1] to value on [xl,xr]
def denormalize ( xi, xl, xr ):
    return 0.5 * ( xr + xl ) + ( xi / 2.0 ) * ( xr - xl )

# The derivative of the denormalize function
def jacobian ( xl, xr ):
    return 0.5 * ( xr - xl )

# Generate a lambda function for a coordinate range for the denormalize function
def fdenormalize ( xl, xr ):
    return lambda xi: denormalize( xi, xl, xr )
```

Description

The code in this script generates a list of shape functions and a list of their derivatives. The shape functions are **Lagrange polynomials** over the domain [-1, 1]. Each shape function will be equal to 1 at the x-coordinate that corresponds to it's index in the shape function array, and zero at all others.

For example, if 3 element nodes are requested, the domain [-1, 1] will be split into three evenly spaced x-values: [-1.0, 0.0, 1.0]. Then, three shape functions are created: [N_0, N_1, N_2]. The result of calling N_0 at x[0] will be 1, and will equal zero at the other two x-values, as shown:

- `N`$_0$` ( x[0] ) = 1.0`
- `N`$_0$` ( x[1] ) = 0.0`
- `N`$_0$` ( x[2] ) = 0.0`

The functions are generated as a summation of lambda functions, which while are probably not the most efficient, provide us with an easy way to generalize the Lagrange polynomial, defined as:

$$N_i(x) = \prod_{j=1(j\neq i)}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x-x_1)(x-x_2)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_1)(x_i-x_2)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)}$$

and it's derivative, defined as:

$$N_i'(x) = \frac{\sum_{k=0}^{n}\prod_{l=0,l\neq k}^{n}(x-x_l)}{\prod_{k=0,k\neq j}^{n}(x_j-x_k)}$$

Code

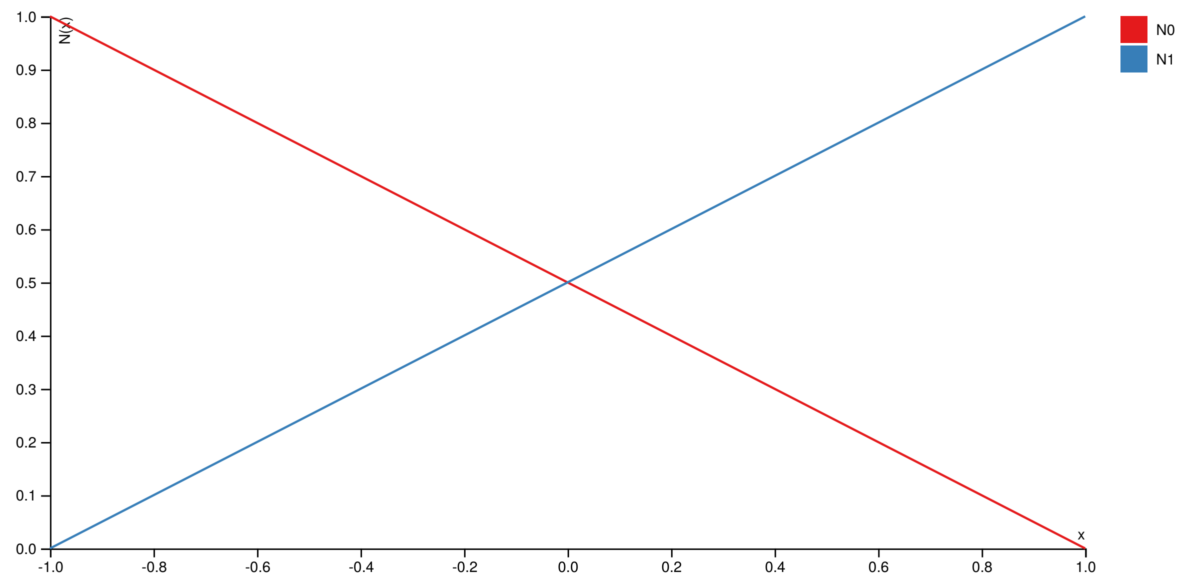
The complete code, shown below, can also be found on **Github**

```
import matplotlib.pyplot as plt

def shape_functions ( num_element_nodes ):

    # Create xi array
    xi = [ -1. + float(i) * 2. / float( num_element_nodes - 1 )  for i in range( num_element_nodes ) ]

    # Create N and dN arrays
    N = []
    dN = []

    for i in range( num_element_nodes ):

        # Create shape function
        f = []

        for j in range( num_element_nodes ):

            if i != j:

                f.append( lambda _xi, _i=xi[i], _j=xi[j]: ( _xi - _j ) / ( _i - _j ) )

        N.append( lambda _xi, _f=f: reduce( lambda a,b: a*b, map( lambda c: c(_xi), _f ) ) )


        # Create derivative of shape function
        mf = []

        for j in range( num_element_nodes ):

            if i != j:

                m = 1.0 / ( xi[ i ] - xi[ j ] )
                f = []

                for k in range( num_element_nodes ):

                    if k != i and k != j:

                        f.append( lambda _xi, _i=xi[ i ], _k=xi[ k ]: ( _xi - _k ) / ( _i - _k ) )

                    else:

                        f.append( lambda _xi: 1.0 )

                mf.append( ( m, f ) )

        dN.append( lambda _xi, _mf=mf: reduce( lambda a,b: a+b, [ _m * reduce( lambda c,d: c*d, map( lambda __f: __f(_xi)

    return N, dN, xi

def plot_shape_functions ( shape_fns ):

    npoints = 250
    x = [ -1 + i * 2. / ( npoints- 1 )  for i in range( npoints ) ]

    for f in shape_fns:
        plt.plot( x, [ f( i ) for i in x ] )

    plt.show()

def plot_f_df ( f, df ):

    npoints = 250
    x = [ -1 + i * 2. / ( npoints- 1 )  for i in range( npoints ) ]

    plt.plot( x, [ f( i ) for i in x ] )
    plt.plot( x, [ df( i ) for i in x ] )
    plt.show()


if __name__ == '__main__':
    N, dN, xi = shape_functions( 3 )
    plot_shape_functions( N )
    # plot_f_df( N[0], dN[0] )
```

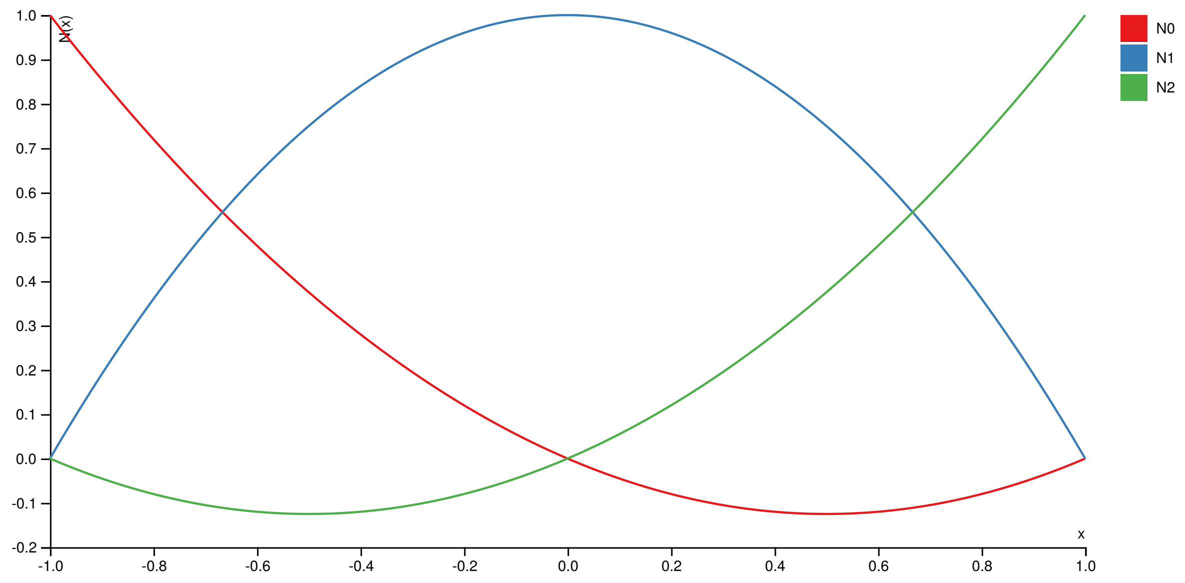
Sample Plots

The following plots show a few examples of the functions that are generated by the script.

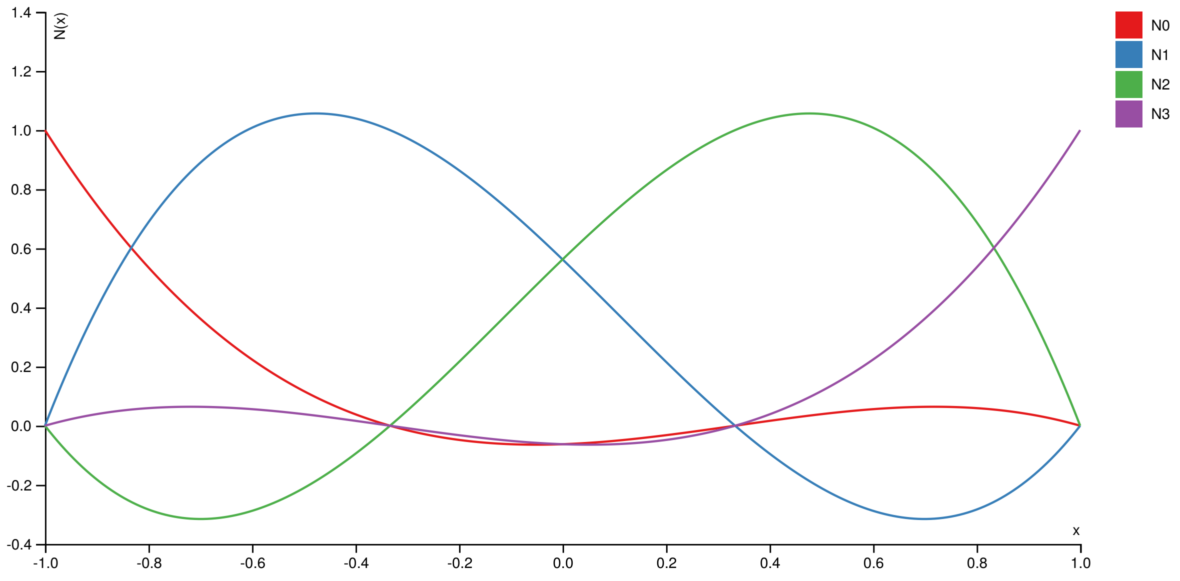
2 element nodes



3 element nodes



4 element nodes


N(x)
x
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-1.0
-0.8
-0.6
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
N0
N1
N2
N3