```
In [1]: import sys
        from scipy import linalg as LA
        from scipy.interpolate import interp1d
        import numpy as np
        from matplotlib import pyplot as plt
        sys.path.append('/Users/Lampe/PyScripts')
        import blfunc as bl
        from IPython.display import display
        from sympy import *
        from sympy import symbols
        from sympy import init_printing
        init_printing()
        np.set_printoptions(precision = 2, suppress = True)
```

Calculate Elemental Stiffness Matrix

```
In [2]: h, zeta, x_global, e = symbols('h zeta xi e')

        # define transformation relation between local and global coordinates:
        # x_global = zeta + (e - 1) * h
        # zeta = local coordinate,
        # e = global element number
        # h = element width
        #
        x_global = (e - 1) * h # defines the starting location for each element in global coordinates
```

```
In [3]: k_e_11_sym = integrate(1/h**2 * ((zeta + x_global)**2 - 1) + 12 * (1 - zeta/h)**2, (zeta, 0, h))
        simplify(k_e_11_sym)
```

Out[3]: $e^2 h - eh + \frac{13h}{3} - \frac{1}{h}$

```
In [4]: k_e_12_sym = integrate((-1/h**2)*((zeta + x_global)**2-1) + 12*(1-zeta/h)*(zeta/h),(zeta, 0, h))
        simplify(k_e_12_sym)
```

Out[4]: $-e^2 h + eh + \frac{5h}{3} + \frac{1}{h}$

```
In [5]: k_e_22_sym = integrate((1/h**2)*((zeta + x_global)**2-1)+12*(zeta/h)**2,(zeta,0,h))
        simplify(k_e_22_sym)
```

Out[5]: $e^2 h - eh + \frac{13h}{3} - \frac{1}{h}$

Define Function for Elemental Stiffness Matrix

```
In [6]: def k_elemental(e, h):
            """ Elemental Stiffness matrix for bilinear element.
            The elemental stiffness matrix is a 4x4 element, having values not equal to zero only over the defined element.
            The transformation from elemental (local) to global coordinates is performed here.  Therefore, only the element
            number is needed.  The local to global transformation was performed using:  x = zeta + (e - 1) * h
            e (element number): must have a minimium value of 1 (unity)
            h (element width):"""
            k_11 = e**2 * h - e * h + 13.0 * h / 3.0 - 1.0 / h
            k_12 = - e**2 * h + e * h + 5.0 * h / 3.0 + 1.0 / h
            k_21 = k_12
            k_22 = e**2 * h - e * h + 13.0 * h / 3.0 - 1.0 / h
            k = np.array([[k_11, k_12],[k_21, k_22]])

            return k
```

b. Find the approximate solutions using piecewise linear elements for different numbers of elements.

In [7]:
```
n_el = np.array([2, 4, 8, 16, 32, 64]) # number of elements
u_0 = 0 # BCT at x = 0
u_1 = 1 # BCT at x = 1
bdry = [0.0, 1.0] # location of BCTs
node_per_el = 2.0 # bilinear element

domain_size = bdry[1] - bdry[0]

# create loop for different discretizations (number of elements per domain)
for m in range(len(n_el)):
    h = domain_size / n_el[m] # element width
    n_node = n_el[m] + 1 # nodes (dof) in domain

    # create empty arrays
    k_global = np.zeros((n_node, n_node))# global stiffness matrix
    f_global = np.zeros(n_node)# global forcing vector
    alpha = np.zeros(n_node)# global solution, equal to the approximate solution () for FEM

    # create global stiffness matrix and forcing vector
    for i in xrange(n_el[m]):
        k_global[i:i + node_per_el, i:i + node_per_el] = k_elemental(i + 1, h) + k_global[i:i + node_per_el, i:i + node_per_el
        f_global[i:i + node_per_el] = 0 # if the problem had Nat. BCTs, they would be accounted for here (in the forcing vecto
)

    # solve for alpha vector (inner - not boundary values)
    bct_0 = u_0 * k_global[:,0] # define Essential boundary condition at u(0)
    bct_1 = u_1 * k_global[:, int(n_node) - 1] # define Essential boundary condition at u(1)
    k_inner = k_global[1:int(n_node)-1, 1:int(n_node) - 1] # define stiffness matrix that does not include Ess. BCTs

    # move Ess. BCTs to rhs and subtract them from the original forcing vector (f_global)
    # these BCTs are effectively forces on the system
    rhs = f_global[1:int(n_node) - 1] - bct_0[1:int(n_node) - 1] - bct_1[1:int(n_node) - 1]

    # solve for the innner (tems not including Ess. BCTs) alpha vector
    # when using FE method, alpha is equal to the actual displacements we are trying to solve
    # i.e., alpha(x) = u_approx (x)
    alpha[1:int(n_node) - 1] = LA.solve(k_inner, rhs)

    # Explicitly apply the Ess. BCTs to the solution (alpha(x) = u_approx(x) = dislplacements) vector
    alpha[0] = u_0
    alpha[int(n_node)-1] = u_1

    #create vectors for plotting
    if n_el[m] == 2:
        alpha_2el = alpha
        k_global_2el = k_global
        x_2el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_2el_func = interp1d(x_2el, alpha_2el, kind = 'linear') #interpolation function
    elif n_el[m] == 4:
        alpha_4el = alpha
        k_global_4el = k_global
        x_4el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_4el_func = interp1d(x_4el, alpha_4el, kind = 'linear') #interpolation function
    elif n_el[m] == 8:
        alpha_8el = alpha
        k_global_8el = k_global
        x_8el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_8el_func = interp1d(x_8el, alpha_8el, kind = 'linear') #interpolation function
    elif n_el[m] == 16:
        alpha_16el = alpha
        k_global_16el = k_global
        x_16el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_16el_func = interp1d(x_16el, alpha_16el, kind = 'linear') #interpolation function
    elif n_el[m] == 32:
        alpha_32el = alpha
        k_global_32el = k_global
        x_32el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_32el_func = interp1d(x_32el, alpha_32el, kind = 'linear') #interpolation function
    elif n_el[m] == 64:
        alpha_64el = alpha
        k_global_64el = k_global
        x_64el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_64el_func = interp1d(x_64el, alpha_64el, kind = 'linear') #interpolation function
```

Calculate Approximate Solutions at Midpoint of Domain (x = 0.5)

```
In [8]: # get values for convergence study
        discrete = [x_2el, x_4el, x_8el, x_16el, x_32el, x_64el]
        approx = [alpha_2el, alpha_4el, alpha_8el, alpha_16el, alpha_32el, alpha_64el]
        u_converge = np.zeros(6)

        for k in xrange(len(discrete)):
            for i, j in enumerate(discrete[k]):
                if j == 0.5:
                    u_converge[k] = approx[k][i] # array of approximate solutions at x = 0.5
```

Calculate Numerical Derivative Between Nodes

```
In [9]: val_count = np.zeros(len(n_el))

        for i in range(len(n_el)):
            val_count[i] = n_el[i] * 2

            slope = np.diff(approx[i]) / np.diff(discrete[i])
            out_slope  = np.zeros(val_count[i])
            out_x = np.zeros(val_count[i])

            index = 0
            x_index = 0
            for k in xrange(len(slope)):
                add = 0
                for j in xrange(2):
                    out_slope[index] = slope[k]
                    out_x[index] = discrete[i][x_index]

                    index = index + 1
                    if x_index == k:
                        x_index = x_index + 1

            #create vectors for plotting
            if i == 0:
                slope_2el = out_slope
                xslope_2el = out_x
            elif i == 1:
                slope_4el = out_slope
                xslope_4el = out_x
            elif i == 2:
                slope_8el = out_slope
                xslope_8el = out_x
            elif i == 3:
                slope_16el = out_slope
                xslope_16el = out_x
            elif i == 4:
                slope_32el = out_slope
                xslope_32el = out_x
            elif i == 5:
                slope_64el = out_slope
                xslope_64el = out_x
```

Calculate the error norms

In [10]:
```
# create vector of calculation points in domain
x = np.linspace(0, 1, 101)

# array of approximate solutions for differing numbers of elements
# u = u(x)
u_x = np.array([[alpha_2el_func(x)],
                [alpha_4el_func(x)],
                [alpha_8el_func(x)],
                [alpha_16el_func(x)],
                [alpha_32el_func(x)],
                [alpha_64el_func(x)]])

# array of approximate solution derivatives, needed for Energy Error Norm
# du/dx
du_dx = np.array([[np.diff(alpha_2el_func(x))/np.diff(x)],
                  [np.diff(alpha_4el_func(x))/np.diff(x)],
                  [np.diff(alpha_8el_func(x))/np.diff(x)],
                  [np.diff(alpha_16el_func(x))/np.diff(x)],
                  [np.diff(alpha_32el_func(x))/np.diff(x)],
                  [np.diff(alpha_64el_func(x))/np.diff(x)]])

#create empty arrays
norm_L2 = np.zeros(5)
norm_Energy = np.zeros(5)

# used for convergence analysis
# h = element width
h = 1.0 / n_el

# nested for loops calculate L2 and Energy norms of approximate solutions
# norms are calculated for the 6 different mesh densities
for k in xrange(5):
    for i in xrange(len(x) - 2):
#         norm_L2[k] = norm_L2[k] + (x[i] - x[i+1]) * (((u_x[k,0,i] + u_x[k,0,i+1])/2) - ((u_x[k+1,0,i] + u_x[k+1,0,i+1])/2))
        L2a = (x[i] - x[i+1]) * (((u_x[k,0,i] - u_x[k+1,0,i] )**2 /2) + ((u_x[k,0,i+1] - u_x[k+1,0,i+1])**2 /2))
        norm_L2[k] = norm_L2[k] + np.absolute(L2a)

        a = ((1 - (x[i] + x[i + 1])/2.0)**2)
        b = ((du_dx[k,0,i] + du_dx[k,0,i+1])/2 - (du_dx[k+1,0,i] + du_dx[k+1,0,i+1])/2)
        c = (((u_x[k,0,i] + u_x[k,0,i+1])/2) - ((u_x[k+1,0,i] + u_x[k+1,0,i+1])/2))
        norm_Energy[k] = norm_Energy[k] + np.absolute((x[i] - x[i+1]) * (a * b**2 + 12*c**2))

norm_L2 = np.sqrt(norm_L2)
norm_Energy = np.sqrt(0.5 * norm_Energy)
print norm_L2
print norm_Energy
print 1/h[1:6]
```

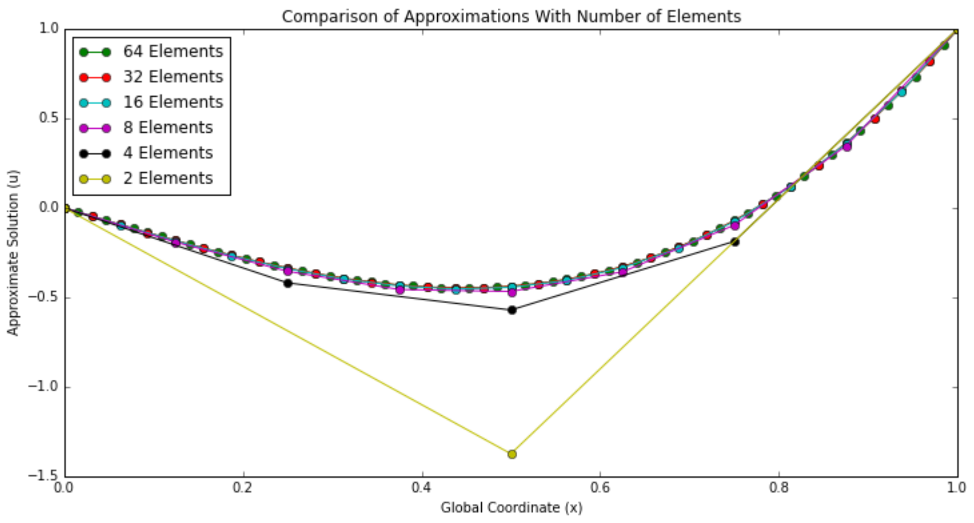
```
[ 0.37  0.05  0.01  0.    0.  ]
[ 1.16  0.2   0.06  0.02  0.01]
[  4.   8.  16.  32.  64.]
```

Plotting

In [13]:
```
from pylab import *
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab

%matplotlib inline
```

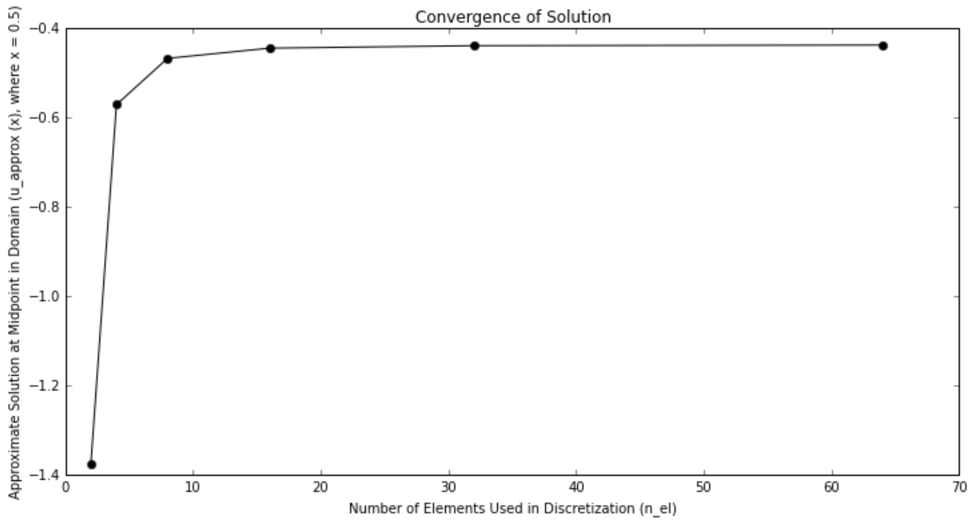
The Approximate Solutions

```
In [14]: fig, ax = plt.subplots(figsize = (12,6))

         ax.plot(x_64el, alpha_64el, 'go-', label="64 Elements")
         ax.plot(x_32el, alpha_32el, 'ro-', label="32 Elements")
         ax.plot(x_16el, alpha_16el, 'co-', label="16 Elements")
         ax.plot(x_8el, alpha_8el, 'mo-', label="8 Elements")
         ax.plot(x_4el, alpha_4el, 'ko-', label="4 Elements")
         ax.plot(x_2el, alpha_2el, 'yo-', label="2 Elements")
         ax.legend(loc=2); # upper left corner
         ax.set_xlabel('Global Coordinate (x)')
         ax.set_ylabel('Approximate Solution (u)')
         ax.set_title('Comparison of Approximations With Number of Elements');
         # fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/HW05/ApproximateSolution.pdf")

         show()
```



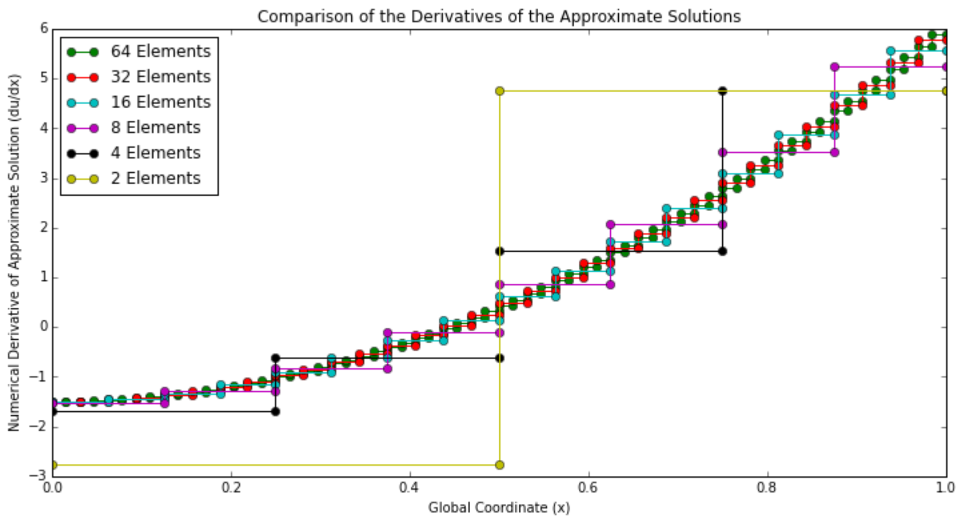
Convergence at x = 0.5

```
In [17]: fig, ax = plt.subplots(figsize = (12,6))

         ax.plot(n_el, u_converge, 'ko-')
         ax.legend(loc=2); # upper left corner
         ax.set_xlabel('Number of Elements Used in Discretization (n_el)')
         ax.set_ylabel('Approximate Solution at Midpoint in Domain (u_approx (x), where x = 0.5)')
         ax.set_title('Convergence of Solution');
         # fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/HW05/Convergence.pdf")

         show()
```



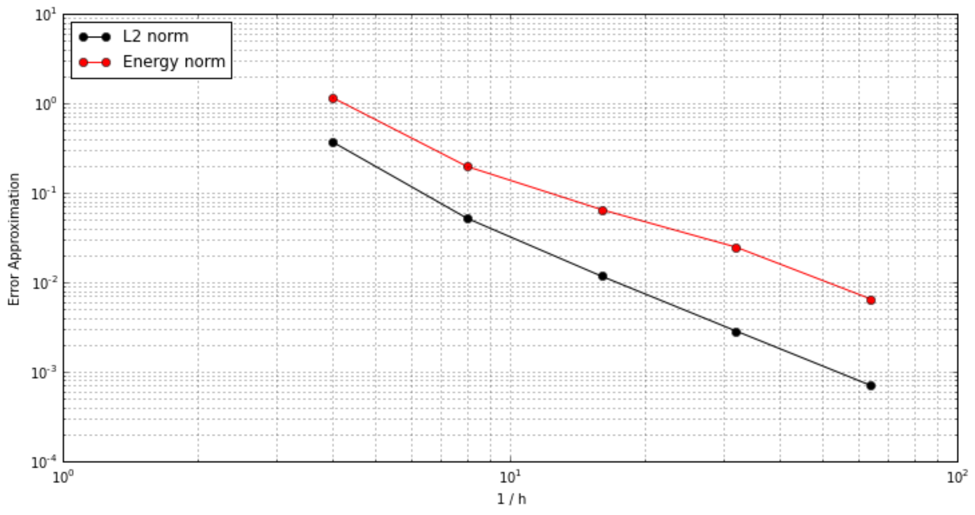
Derivatives (Numerical) of Approximate Solution Across Domain

```
In [18]: fig, ax = plt.subplots(figsize = (12,6))

         ax.plot(xslope_64el, slope_64el, 'go-', label="64 Elements")
         ax.plot(xslope_32el, slope_32el, 'ro-', label="32 Elements")
         ax.plot(xslope_16el, slope_16el, 'co-', label="16 Elements")
         ax.plot(xslope_8el, slope_8el, 'mo-', label="8 Elements")
         ax.plot(xslope_4el, slope_4el, 'ko-', label="4 Elements")
         ax.plot(xslope_2el, slope_2el, 'yo-', label="2 Elements")
         ax.legend(loc=2); # upper left corner
         ax.set_xlabel('Global Coordinate (x)')
         ax.set_ylabel('Numerical Derivative of Approximate Solution (du/dx)')
         ax.set_title('Comparison of the Derivatives of the Approximate Solutions');
         # fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/HW05/ApproximateSolutionDerivative.pdf")

         show()
```



Relative Error Analysis

```
In [20]: fig, ax = plt.subplots(figsize = (12,6))

         ax.plot(1/h[1:6], norm_L2, 'ko-', label="L2 norm")
         ax.plot(1/h[1:6], norm_Energy, 'ro-', label="Energy norm")
         ax.legend(loc=2); # upper left corner
         ax.set_yscale('log')
         ax.set_xscale('log')
         ax.set_xlabel('1 / h')
         ax.set_ylabel('Error Approximation')
         ax.grid(b = True, which = 'minor')
         ax.grid(b = True, which = 'major')
         # ax.set_title('Comparison of Approximations With Number of Elements');
         # fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/HW05/ErrorNorms.pdf")

         show()
```


L2 norm
Energy norm
Error Approximation
1 / h