In [1]:
```
import sys
from scipy import linalg as LA
from scipy.interpolate import interpld
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)
```

In [2]:
```
##################################
# printing functions
##################################
def valprint(string, value):
    """ Inforces uniform formatting of scalar value outputs."""
    print("{0:>15}: {1: .2e}".format(string, value))

def matprint(string, value):
    """ inforces uniform formatting of matrix value outputs."""
    print("{0}:".format(string))
    print(value)
```

Calculate Elemental Stiffness Matrix and Forcing Vector

In [3]:
```
h, zeta, x_global, e, psi_1, psi_1p, psi_2, psi_2p = symbols('h zeta x e psi_1 psi_1p psi_2 psi_2p')

# 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 = zeta + (e - 1) * h # defines the starting location for each element in global coordinates
psi_1 = 1-(zeta/h)
psi_1p = -1/h
psi_2 = zeta/h
psi_2p = 1/h
```

In [4]:
```
k_e_11_sym = integrate(psi_1p * psi_1p * sin(x_global) + psi_1 * psi_1 * cos(x_global), (zeta, 0, h))
simplify(k_e_11_sym)
```

Out[4]:
$$\frac{1}{h^2}\left(-h^2\sin\left(h\left(e-1\right)\right)+2h\cos\left(h\left(e-1\right)\right)-2\sin\left(eh\right)+2\sin\left(h\left(e-1\right)\right)-\cos\left(eh\right)+\cos\left(h\left(e-1\right)\right)\right)$$

In [5]:
```
k_e_12_sym = integrate(psi_1p * psi_2p * sin(x_global) + psi_1 * psi_2 * cos(x_global), (zeta, 0, h))
simplify(k_e_12_sym)
```

Out[5]:
$$\frac{1}{h^2}\left(-h\cos\left(eh\right)-h\cos\left(h\left(e-1\right)\right)+2\sin\left(eh\right)-2\sin\left(h\left(e-1\right)\right)+\cos\left(eh\right)-\cos\left(h\left(e-1\right)\right)\right)$$

In [6]:
```
k_e_22_sym = integrate(psi_2p * psi_2p * sin(x_global) + psi_2 * psi_2 * cos(x_global), (zeta, 0, h))
simplify(k_e_22_sym)
```

Out[6]:
$$\frac{1}{h^2}\left(h^2\sin\left(eh\right)+2h\cos\left(eh\right)-2\sin\left(eh\right)+2\sin\left(h\left(e-1\right)\right)-\cos\left(eh\right)+\cos\left(h\left(e-1\right)\right)\right)$$

In [7]:
```
f_e_1_sym = integrate(psi_1 * x_global, (zeta, 0, h))
simplify(f_e_1_sym)
```

Out[7]:
$$\frac{h^2}{6}(3e-2)$$

In [8]:
```
f_e_2_sym = integrate(psi_2 * x_global, (zeta, 0, h))
simplify(f_e_2_sym)
```

Out[8]:
$$\frac{h^2}{6}(3e-1)$$

Define Function for Elemental Stiffness Matrix

In [9]:
```
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 = (h/3.0)*np.cos(e*h - h) + (1.0/h) * np.sin(e*h - h)
#     k_12 = (h/6.0)*np.cos(e*h - h) + (1.0/h) * np.sin(e*h - h)
#     k_21 = k_12
#     k_22 = (h/3.0)*np.cos(e*h - h) + (1.0/h) * np.sin(e*h - h)

    k_11 = (1/h**2)*(-h**2*np.sin(h*(e-1))+2*h*np.cos(h*(e-1))-2*np.sin(e*h)+2*np.sin(h*(e-1))-np.cos(e*h)+np.cos(h*(e-1)))
    k_12 = (1/h**2)*(-h*np.cos(e*h)-h*np.cos(h*(e-1))+2*np.sin(e*h)-2*np.sin(h*(e-1))+np.cos(e*h)-np.cos(h*(e-1)))
    k_21 = k_12
    k_22 = (1/h**2)*(h**2*np.sin(e*h)+2*h*np.cos(e*h)-2*np.sin(e*h)+2*np.sin(h*(e-1))-np.cos(e*h)+np.cos(h*(e-1)))
    k = np.array([[k_11, k_12],[k_21, k_22]])

    return k

def f_elemental(e, h):
    f_1 = h**2 / 6.0 * (3*e - 2)
    f_2 = h**2 / 6.0 * (3*e - 1)
    f = np.array([f_1, f_2])

    return f
```

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

In [28]:
```
n_el = np.array([2, 4, 8, 16, 32, 64, 128, 256]) # number of elements
u_0 = 1 # 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

#     valprint("element size", h)
#     valprint("nodes in domain", n_node)

    # 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] = f_elemental(i + 1, h) + f_global[i:i + node_per_el]

#     matprint("global Stiffness", k_global)
#     matprint("global forcing vector", f_global)

    # 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
#     matprint("alpha", alpha)

    #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
    elif n_el[m] == 128:
        alpha_128el = alpha
        k_global_128el = k_global
        x_128el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_128el_func = interp1d(x_128el, alpha_128el, kind = 'linear') #interpolation function
    elif n_el[m] == 256:
        alpha_256el = alpha
        k_global_256el = k_global
        x_256el = np.linspace(bdry[0], bdry[1], n_el[m] + 1)
        alpha_256el_func = interp1d(x_256el, alpha_256el, kind = 'linear') #interpolation function
```

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

In [29]:
```
# get values for convergence study
discrete = [x_2el, x_4el, x_8el, x_16el, x_32el, x_64el, x_128el, x_256el]
approx = [alpha_2el, alpha_4el, alpha_8el, alpha_16el, alpha_32el, alpha_64el, alpha_128el, alpha_256el]
u_converge = np.zeros(8)

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 [30]:
```
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
    elif i == 6:
        slope_128el = out_slope
        xslope_128el = out_x
    elif i == 7:
        slope_256el = out_slope
        xslope_256el = out_x
```

Calculate the error norms

In [38]:
```
# 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)],
                [alpha_128el_func(x)],
                [alpha_256el_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)],
                  [np.diff(alpha_128el_func(x))/np.diff(x)],
                  [np.diff(alpha_256el_func(x))/np.diff(x)]])

#create empty arrays
norm_L2 = np.zeros(7)
norm_Energy = np.zeros(7)

# 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(7):
    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 = (np.sin(x[i]) + np.sin(x[i + 1]))
        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 = (np.cos(x[i]) + np.cos(x[i + 1]))
        d = (((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 + c*d**2))

norm_L2 = np.sqrt(norm_L2)
norm_Energy = np.sqrt(0.5 * norm_Energy)
print norm_L2
print norm_Energy
print 1/h[1:8]
```

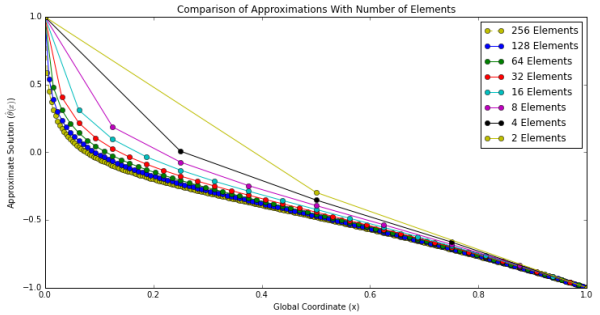
```
[ 0.15  0.11  0.07  0.05  0.04  0.03  0.02]
[ 0.44  0.37  0.29  0.21  0.12  0.08  0.05]
[   4.    8.   16.   32.   64.  128.  256.]
```

Plotting

In [32]:
```
from pylab import *
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab

%matplotlib inline
```

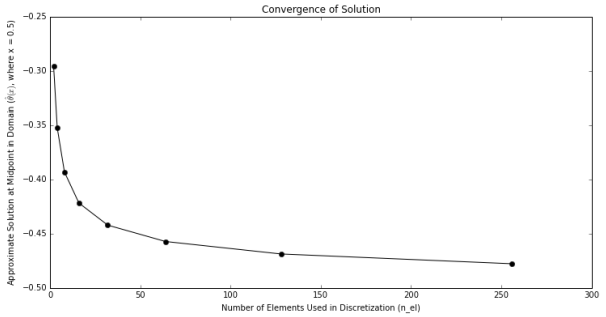
The Approximate Solutions

In [33]:
```
fig, ax = plt.subplots(figsize = (12,6))

ax.plot(x_256el, alpha_256el, 'yo--', label="256 Elements")
ax.plot(x_128el, alpha_128el, 'bo-', label="128 Elements")
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=1); # upper left corner
ax.set_xlabel('Global Coordinate (x)')
ax.set_ylabel('Approximate Solution ('r'$\hat{\theta} (x)$)')
ax.set_title('Comparison of Approximations With Number of Elements');
fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/TakeHomeExam/ApproximateSolution.pdf")

show()
```



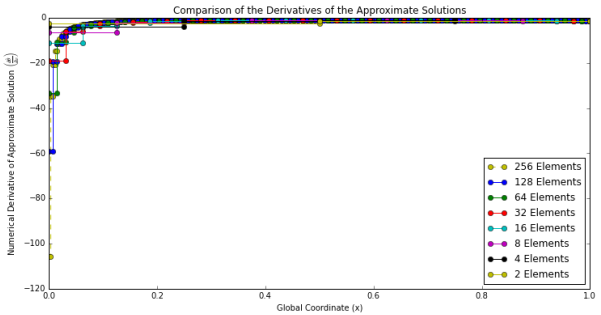
Convergence at x = 0.5

In [34]:
```
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 ('r'$\hat{\theta} (x)$, where x = 0.5)')
ax.set_title('Convergence of Solution');
fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/TakeHomeExam/Convergence.pdf")

show()
```

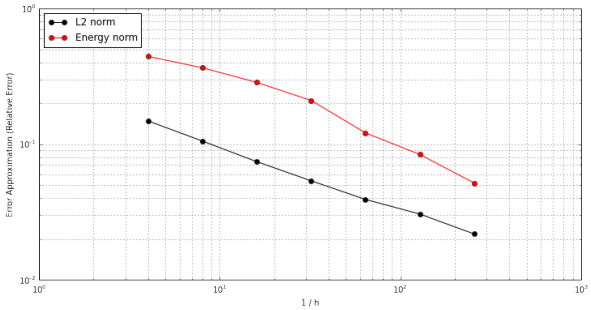


Derivatives (Numerical) of Approximate Solution Across Domain

In [44]:
```
fig, ax = plt.subplots(figsize = (12,6))

# ax.set_yscale('log')
ax.plot(xslope_256el, slope_256el, 'yo--', label="256 Elements")
ax.plot(xslope_128el, slope_128el, 'bo-', label="128 Elements")
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=4); # upper left corner
ax.set_xlabel('Global Coordinate (x)')
ax.set_ylabel('Numerical Derivative of Approximate Solution r'$\left(\frac{\partial \hat{\theta}}{\partial x}\right)$')
ax.set_title('Comparison of the Derivatives of the Approximate Solutions');
fig.savefig("/Users/Lampe/Documents/UNM_Courses/ME-504_ComputationalMechanics_Brake/TakeHomeExam/ApproximateSolutionDerivative
pdf")

show()
```



In [41]:
```
fig, ax = plt.subplots(figsize = (12,6))

ax.plot(1/h[1:8], norm_L2, 'ko-', label="L2 norm")
ax.plot(1/h[1:8], 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 (Relative Error)')
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/TakeHomeExam/ErrorNorms.pdf")

show()
```



In []:

Relative Error Analysis