HW05_prob2

November 25, 2015

General Class for the 1D Analysis of the Heat Eqn

In [1]:
```
from scipy.sparse import diags
import scipy.linalg
import numpy as np

class OneDim_SS_HeatEqn_FD():

    def __init__(self, nodes, k=lambda x: 1, Q=lambda x:0, T0=0, dTL=0, area = 1):
        """
            Initiates an object from the argument list to perform FD solution of the steday-sta
            Primary variable: T
            Independent variable: x

            **Governing Equation:
            \frac{d}{dx}\left(k^A \frac{dT}{dx}\right) + Q(x) = 0

            * **Problem Domain**  £ 0\le x \le L £

            * **Boundary Conditions**
                * Dirichlet: £T(0) = T_0£
                * Nuemann: £ \left(k^A \frac{dT}{dx} \right)_{x=L} = Q_L £

            Input Arguments:
            (required) nodes: 1D array of nodal locations
            (optional) k : coeficient function of x (default = 1)
            (optional) Q : forcing function of x (default = 0)
            (optional) T0: temperature (essential) BCT at x = 0 (default = 0)
            (optional) dTL: derivative of temperature (natural => dTL = nQ/kA) BCT at x = L (de
                where n = outward normal, A = cross-sectional area, Q = flux, k = thermal cond.
            (optional) area: cross-sectional area of domain, orthogonal to direction of heat fl

            Output:
            T at the locations defined by the nodes input array
        """
        self.nodes = np.array(nodes, dtype=np.double)
        self.k = k
        self.Q = Q
        self.T0 = T0
        self.dTL = dTL
        self.area = area
        self.h = (max(nodes) - min(nodes)) / (np.double(len(nodes)) - 1) # element size
        self.node_cnt = len(nodes)
```

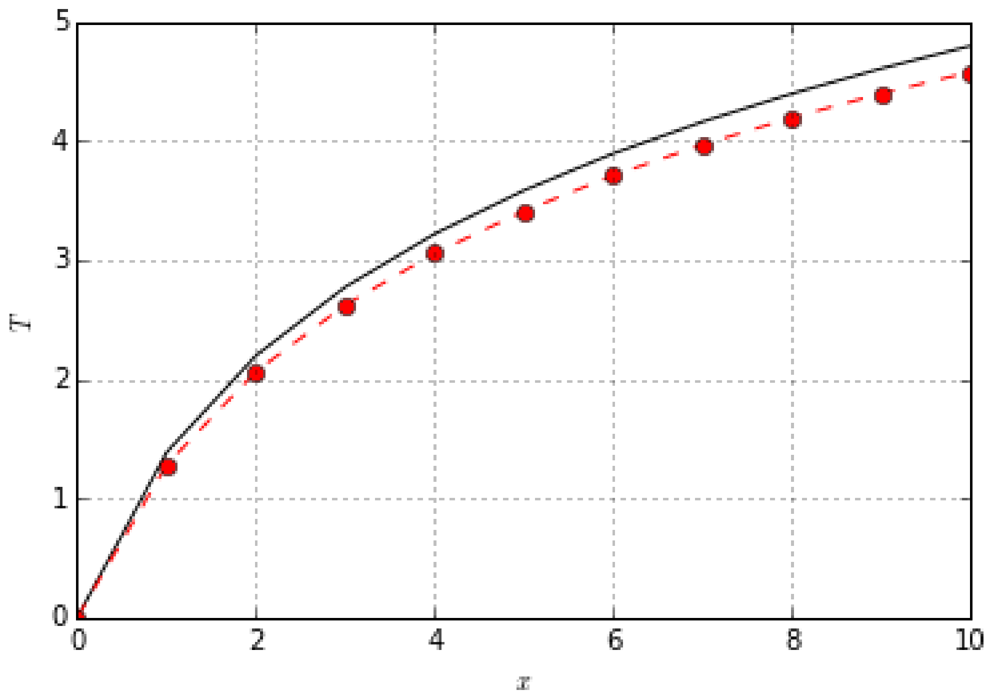
```python
    def kA(self):
        """
            returns thermal conductive \times area (kA) and flux at x = L
        """
        self.kA = self.k * self.area
        self.flux = self.kA * self.dTL
        return self.kA, self.flux

    def apply_bcs(self):
        """
            modify [K] and {F} to account for boundary conditions
        """
        self.K = diags([-1,2,-1],[-1,0,1], shape=(self.node_cnt,self.node_cnt)).toarray() # int
        self.K[0,0:2] = np.array([1, 0]) # modifies first equation
        self.K[self.node_cnt-1,self.node_cnt-2:self.node_cnt] = np.array([-1,1]) # modifies las
        self.K = kA/np.double(self.h)**2 * self.K #accounts for thermal cond & element spacing

        self.F = self.Q(nodes) # interior nodes
        self.F[0] = kA/np.double(self.h)**2 * self.T0 # modifies first equation
        self.F[-1] = kA/np.double(self.h) * self.dTL # modifies last equation
        return self.K, self.F

    def solve(self):
        self.apply_bcs()
        return scipy.linalg.solve(self.K, self.F)
```

Check that algorithm is working

In [2]:
```python
import numpy as np
import math

#define problem discretization
n = 11 # number of nodes
x0 =0 # left boundary
xL = 10 # right boundary
nodes = np.linspace(x0, xL, n)

#constants
k = 2 # thermal conductivity
area = 10 # cross-sectional area perpendicular to heat flow
kA = k*area
T_const = 2

#boundary terms
T_0 = T_const * np.log(1) # essential BCT at x=0
dT_L = T_const / np.double((xL + 1))# natural BCT at x=L, flux*n = kA*dT/dx

#forcing function
Q = lambda x, coef = kA: coef * T_const/(x+1)**2 # the forcing function Q(x)
trial_01 = OneDim_SS_HeatEqn_FD(nodes, k, Q, T_0, dT_L, area) # cont -> optional argument
out = trial_01.solve()
```

Compare with Manufactured Solution

```
In [3]: %matplotlib inline
        import matplotlib.pyplot as plt

        T_an = map(lambda x: T_const * math.log(x+1), nodes)
        plt.plot(nodes, T_an, 'k-', nodes, out, 'ro--');

        plt.xlabel('$x$')
        plt.ylabel('$T$')
        plt.grid()
```



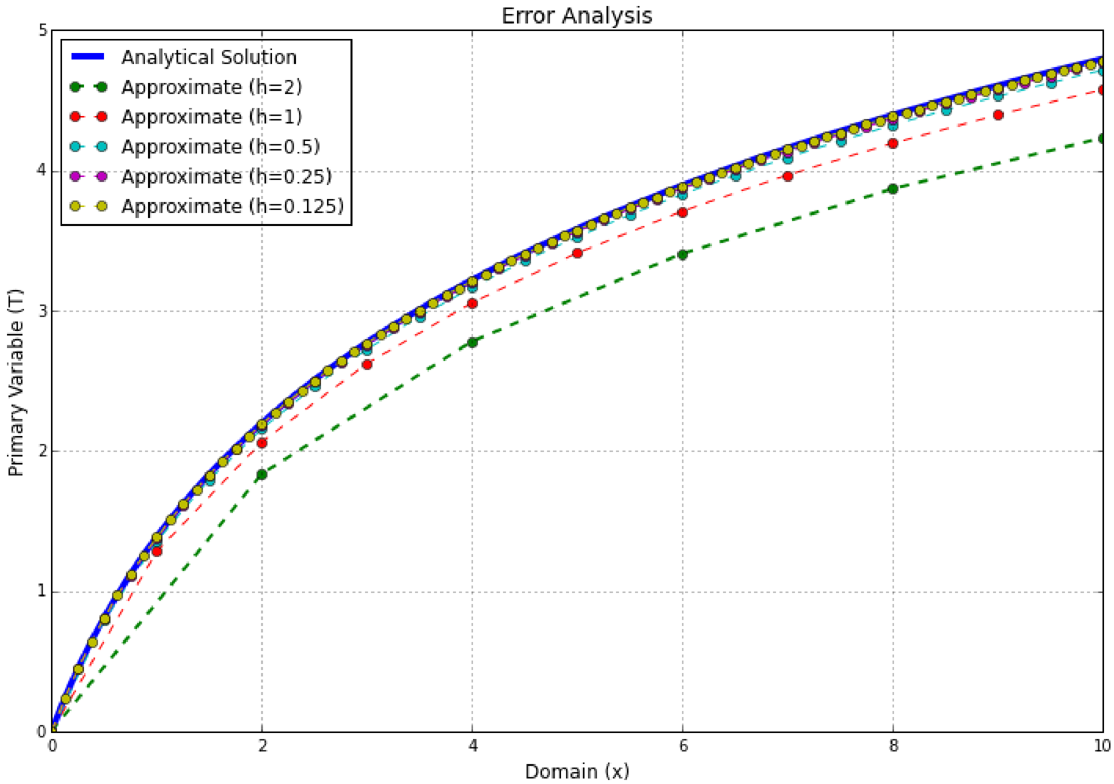
Analyze Rate of Convergence

```
In [4]: n = np.array([6, 11, 21, 41, 81]) #number of nodes
        out_array = np.zeros((max(n), len(n)))

        # domain
        x0 =0 # left boundary
        xL = 10. # right boundary
        h = (xL - x0)/(n-1.)

        #constants
        k = 2 # thermal conductivity
        area = 10 # cross-sectional area perpendicular to heat flow
        kA = k*area
        T_const = 2

        #boundary terms
        T_0 = T_const * np.log(1) # essential BCT at x=0
```

```
dT_L = T_const / np.double((xL + 1))# natural BCT at x=L, flux*n = kA*dT/dx

#forcing function
Q = lambda x, coef = kA: coef * T_const/(x+1)**2 # the forcing function Q(x)

j = 0
for i in n:
    nodes = np.linspace(x0, xL, n[j])
    problem = OneDim_SS_HeatEqn_FD(nodes, k, Q, T_0, dT_L, area)
    out_array[0:i,j] = problem.solve()
    j = j+1
```

Plot Results

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import math

# manufactured solution
nodes_an = np.linspace(x0, xL, 81)
T_an = map(lambda x: T_const * math.log(x+1),nodes_an )

fig_r, ax = plt.subplots(figsize = (12,8))

ax.plot(nodes_an, T_an, '-', markersize=10, lw = 4, label='Analytical Solution')
ax.plot(np.linspace(x0, xL, n[0]), out_array[0:n[0],0], 'o--', markersize=6, lw = 2, label='App
ax.plot(np.linspace(x0, xL, n[1]), out_array[0:n[1],1], 'o--', markersize=6, lw = 1, label='App
ax.plot(np.linspace(x0, xL, n[2]), out_array[0:n[2],2], 'o--', markersize=6, lw = 1, label='App
ax.plot(np.linspace(x0, xL, n[3]), out_array[0:n[3],3], 'o--', markersize=6, lw = 1, label='App
ax.plot(np.linspace(x0, xL, n[4]), out_array[0:n[4],4], 'o--', markersize=6, lw = 1, label='App

ax.set_xlabel('Domain (x)', fontsize = 12)
ax.set_ylabel('Primary Variable (T)', fontsize = 12)
ax.set_title('Error Analysis' , fontsize = 14)
ax.grid(b = True, which = 'major')
ax.grid(b = True, which = 'major')
# ax.set_ylim(-2, 2)
# ax.set_xlim(0,4)
# ax.set_xscale('log')
# ax.set_yscale('log')
ax.legend(loc=0)

fig_name = '2_Results.pdf'
path = '/Users/Lampe/Documents/UNM_Courses/ME-500/HW05/'
fig_r.savefig(path + fig_name)
# show()
```



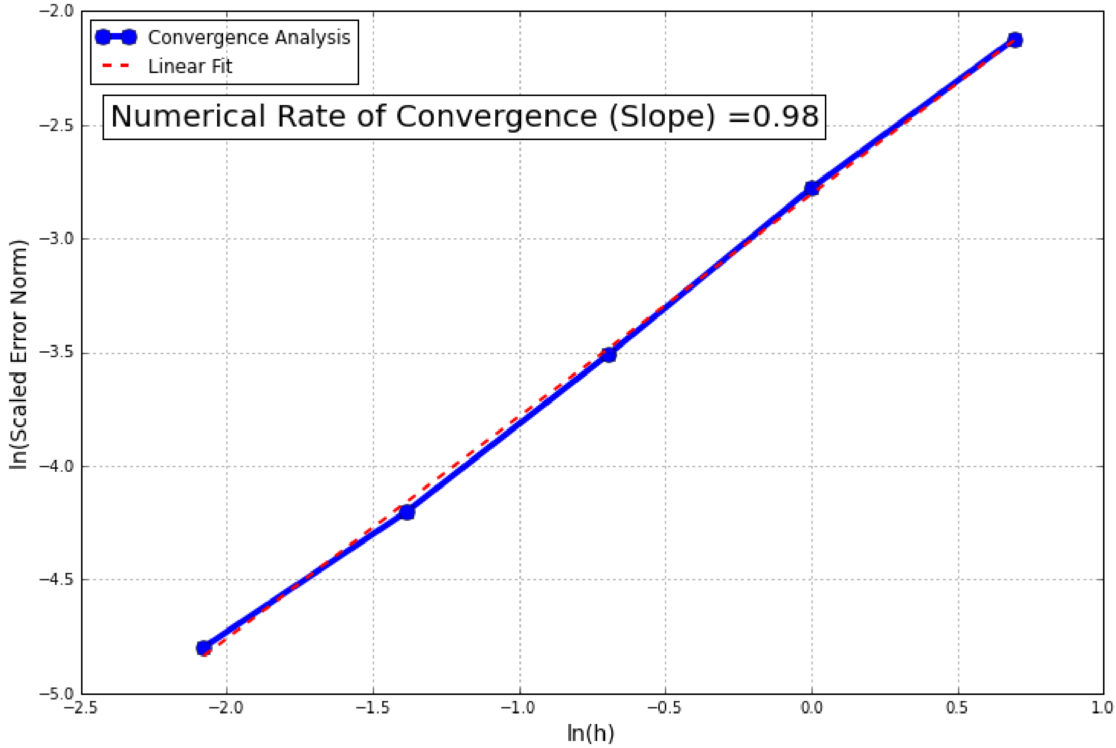
Calculate Error for Each Discretization

In [6]:
```
nodes_an.shape
h_an = (xL - x0)/(nodes_an.shape[0]-1)
p = 2# p2 norm
scaled_error_norm = np.zeros(len(n))

for i in xrange(len(n)):
    el_size_out = nodes_an[h[i]/h_an] # locations of numerical results in the domain
    idx = el_size_out / h_an # index to identify increment of analytical results
    sol_ann = T_an[::int(idx)] # analytical results at node locations of numerical results
    sol_num = out_array[0:n[i],i] # numerical results
    error = np.abs(sol_ann - sol_num) # error vector
    scaled_error_norm[i] = 1./len(out_array)**(1./p) * np.sum(np.abs(error)**p)**(1./p) # scale
```

Plot Numerical Rate Convergence

In [9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import math

log_h = np.log(h)
log_scaled_error_norm = np.log(scaled_error_norm)

# fit to data
m, b = np.polyfit(log_h, log_scaled_error_norm, 1)
```

```python
# create figure
fig_conv, ax = plt.subplots(figsize = (12,8))

ax.plot(log_h, log_scaled_error_norm, '-o', markersize=10, lw = 4, label = 'Convergence Analysis
ax.plot(log_h, m * log_h + b, '--r',lw = 2, label = 'Linear Fit')

# annotate plots with text boxes
lbl = ''r'Numerical Rate of Convergence (Slope) ={:.2f}'.format(m)
ax.text(-2.4, -2.5, lbl, bbox={'facecolor':'white', 'pad':10}, fontsize = 20)

ax.set_xlabel('ln(h)', fontsize = 14)
ax.set_ylabel('ln(Scaled Error Norm)', fontsize = 14)
# ax.set_title('Error Analysis' , fontsize = 14)
ax.grid(b = True, which = 'major')
ax.grid(b = True, which = 'major')
# ax.set_ylim(-2, 2)
# ax.set_xlim(0,4)
# ax.set_xscale('log')
# ax.set_yscale('log')
ax.legend(loc=0)

fig_name = '2_Converg.pdf'
path = '/Users/Lampe/Documents/UNM_Courses/ME-500/HW05/'
fig_conv.savefig(path + fig_name)
# show()
```



In [ ]: