HW05_prob4

November 25, 2015

General Class for the 1D Analysis of the Heat Eqn

In [1]:
```
import ipdb
```

In [24]:
```
from scipy.sparse import diags
import scipy.linalg
import numpy as np
np.set_printoptions(precision=2, threshold=1000, suppress=False, linewidth=80)

class OneDim_Trans_HeatEqn_FD():

    def __init__(self, nodes, times, IC, C=lambda x: 1, k=lambda x: 1, f=lambda x:0,
                 BC_type = [1, 0], BC_0=1, BC_L=0, area = 1, alpha = 0.5):
        """
            Initiates an object from the argument list to perform FD solution of the steday-st
            Primary variable: T
            Independent variables: x & t

            **Governing Equation:
            C \frac{dT}{dt} - \frac{d}{dx}\left(k^A \frac{dT}{dx}\right) = f(x)

            * **Problem Domain**  £ x0\le x \le xL \quad & t0\le t \le tf£

            Input Arguments:
            (required) nodes: 1D array of nodal locations
            (required) times: 1D array of
            (required) IC: intial value of primary variable, these values may be overwritten a
            (optional) C : capacitance, may be a function of x (default = 1)
            (optional) k : coeficient function of x (default = 1)
            (optional) f : forcing function of x (default = 0)
            (optional) BC_type: tuple for defining boundary condition types -> [BC @ x = 0, BC
                Dirichlete (essential)
                    => 1, place "1" for constant BC
                    => 2, place "2" for BC that is a function of time
                    temperature prescribed (default = 0)
                Neumann (natural) => 0, place "0" for this type of BC
                    flux prescribed : Q = nkA (dT/dx)
                    where n = outward normal, A = cross-sectional area, Q = flux, k = thermal
                e.g., essential BC at x=0 and neumann at x=L -> BC_type = [1,0]
            (optional) BC_0: value of prescribed temperature or flux at x = 0
            (optional) BC_L: value of prescribed temperature or flux at x = L
            (optional) area: cross-sectional area of domain, orthogonal to direction of heat f
            (optional) alpha: defines the numerical integraction method (general trapezoidal r
                alpha = 0 => explicit integration
```

```
            alpha = 0.5 => Implicit integration (highest rate of convergence) (default)
            alpha = 1 => fully implicit (most stable for dynamic problems)

        Output:
        T_sol: 2D array of the numerical approximation at defined nodes and points in time
            - each row represents the spatial solution at a point in time, T_sol[8,:] = so
                all nodes during the 8th time step
            - each column reprsents temporal solution at a node, T_sol[:,3] = solution at
                for all times

        Example Input:
            example = OneDim_Trans_HeatEqn_FD(nodes,t_arr,IC, C, k, F, bc_type, T_0, T_L,
            The solution will be stored in the Class "example" and the solution is obtaine
            the "solve" method: example.solve()

    """
    self.nodes = np.array(nodes, dtype=np.double) # spatial discretization
    self.t = np.array(times, dtype=np.double) # temporal discretization
    self.k = k # thermal conductivity
    self.f = f # forcing function

    self.IC = np.array(IC, dtype=np.double) # initial conditions
    self.alpha = alpha #defines integration method

    self.BC_type = BC_type # defines type of BCT: 1=> Dirichlete, !=1 => Flux (Neumann)
    self.BC_0 = BC_0 # magnitude of BCT at x = 0
    self.BC_L = BC_L # magnitude of BCT at x = L
    self.area = area # x-sectional area perpendicular to x
    self.h = (max(nodes) - min(nodes)) / (np.double(len(nodes)) - 1) # element size
    self.node_cnt = len(nodes)

def time_step(self):
    if len(self.t) > 1:
        self.s = self.t[1] - self.t[0] # time step size
    else:
        self.s = 1
    return self.s

def kA(self):
    """
        returns thermal conductivity multiplied by area (kA)
    """
    self.kA = self.k * self.area
    return self.kA

def assemble_C(self):
    """
        builds diagonal capacitance matrix [C]
        - the nodes for x=0 and x=L will be modified for BCTs
    """
    self.C = C * diags([1],[0],shape=(self.node_cnt,self.node_cnt)).toarray()# capacitance
    return self.C

def assemble_K(self):
```

```
        """
            builds stiffness matrix [K]
            - the nodes for x=0 and x=L will be modified for BCTs
            - interior nodes will not be modified further
        """
        self.K = diags([-1,2,-1],[-1,0,1], shape=(self.node_cnt,self.node_cnt)).toarray() # in
        self.K = kA/np.double(self.h)**2 * self.K #accounts for thermal cond & element spacing
        return self.K


    def apply_bc_A(self):
        """
            account for boundary conditions at x = 0 & x = L
            - additionally, the stiffness matrix [K] is modified to maintain symmetry => posit
        """
        # apply BC at x = 0
        if self.BC_type[0] != 0:
            # essential BC
            self.A[0,0:2] = np.array([1, 0]) # modifies first equation
            self.A[1,0] = 0 # modification to mainatin symmetry
        elif self.BC_type[0] == 0:
            # natural BC, Flux (Q*)
            n = -1.0 # unit outward normal
            dT = self.BC_0 / (kA * n)
            self.A[0,0:2] = kA/np.double(self.h)**2 * np.array([1, -1]) # modifies [K] first e

        # apply BC at x = L
        if self.BC_type[1] != 0:
            # essential BC
            self.A[-1][-2:] = np.array([0, 1]) # modifies last equation
            self.A[-2][-1] = 0# modification to mainatin symmetry
        elif self.BC_type[1] == 0:
            # natural BC, flux (Q*)
            n = 1.0 # unit outward normal
            dT = self.BC_L / (kA * n)
            self.A[-1][-2:] = kA/np.double(self.h)**2 * np.array([-1, 1]) # modifies last equa
        return self.A

    def apply_IC(self):
        """
            account for initial conditions at (boundaries) x = 0 & x = L
        """
        # force initial condition to satisfy BC
        # boundary condtion at x = 0
        if self.BC_type[0] == 1:
            # essential BC - constant
            self.IC[0] = self.BC_0
        elif self.BC_type[0] == 2:
            # essential BC - time dependent
            self.IC[0] = self.BC_0(self.t[0])

        # boundary condtion at x = L
        if self.BC_type[1] == 1:
            # essential BC - constant
```

```
            self.IC[-1] = self.BC_L
        elif self.BC_type[1] == 2:
            # essential BC - time dependent
            self.IC[-1] = self.BC_L(self.t[0])
        return self.IC

    def apply_bc_b(self, t=0):
        """
            account for boundary conditions in the {b} vector at x = 0 & x = L
            - additionally, modifications made to maintain symmetry of [A] => positive definit
        """
        # boundary condtion at x = 0
        if self.BC_type[0] == 1:
            # essential BC - constant
            self.b[0] = self.BC_0 # modifies first equation
            self.b[1] = self.b[1] - self.A_old[0] * self.BC_0# modification to mainatin symmet
        elif self.BC_type[0] == 2:
            # essential BC - time dependent
            self.b[0] = self.BC_0(t) # modifies first equation
            self.b[1] = self.b[1] - self.A_old[0] * self.BC_0(t) # modification to mainatin sy
        elif self.BC_type[0] == 0:
            # natural BC, Flux (Q*)
            n = -1.0 # unit outward normal
            dT = self.BC_0 / (kA * n)
            self.b[0] = kA/np.double(self.h)**2 * (-kA/np.double(self.h)) * dT # modifies {F}

        # boundary condition at x = L
        if self.BC_type[1] == 1:
            # essential BC - constant
            self.b[-1] = self.BC_L # modifies last equation
            self.b[-2] = self.b[-2] - self.A_old[1] * self.BC_L # modification to mainatin sym
        elif self.BC_type[1] == 2:
            # essential BC - time dependent
            self.b[-1] = self.BC_L(t) # modifies last equation
            self.b[-2] = self.b[-2] - self.A_old[1] * self.BC_L(t) # modification to mainatin
        elif self.BC_type[1] == 0:
            # natural BC, flux (Q*)
            n = 1.0 # unit outward normal
            dT = self.BC_L / (kA * n)
            self.b[-1] = kA/np.double(self.h)**2 * kA/np.double(self.h) * dT # modifies last e
        return self.b

    def solve(self):
        """
            Main function where the transient problem is solve
        """
        self.time_step() # call in step size (uniform)
        self.assemble_C() # call in capacitance matrix
        self.assemble_K() # call in stiffness matrix
        T_sol = np.zeros((len(self.t),len(self.nodes))) # build solution array

#         ipdb.set_trace()
        self.A = self.C + self.alpha * self.s * self.K # build [A]
        self.A_old = np.array([self.A[1,0], self.A[-2][-1]]) # components to enforce symm., us
```

```
            self.B = self.C - (1 - self.alpha) * self.s * self.K # build [B]

            # forcing function is assumed constant in time
            self.F = self.alpha * self.s * self.f(nodes) + (1 - self.alpha) * self.s * self.f(nodes
            self.apply_bc_A() # make modifications to [A] and intial conditions for boundary condi
            self.apply_IC() # enforce initial conditions to match BCTs

            T_old = self.IC # current time
            T_sol[0,:] = T_old # assign first row from initial conditions

            Aq, Ar = scipy.linalg.qr(self.A) #decompose [A], Aq -> orthogonal, Ar -> upper triangu

            for i in xrange(len(self.t[1:])):
                # solve: [A]{T_new} = {b} for all times
                t = self.t[i+1]
                self.b = np.dot(self.B, T_old) + F(nodes)
                self.apply_bc_b(t) # apply boundary conditions to {b}
                b_hat = np.transpose(Aq).dot(self.b) # [Q]^inv {b} = b_hat
                T_new = scipy.linalg.solve_triangular(Ar, b_hat) # performs back substitution
#                 ipdb.set_trace()
                T_sol[i+1,:] = T_new
                T_old = T_new
            return T_sol
```

Problem 4.i Setup problem data

In [34]:
```
#define problem discretization
n = 21 # number of nodes
x0 =0 # left boundary
xL = 1 # right boundary
nodes = np.linspace(x0, xL, n)

#constants
k = 2 # thermal conductivity
area = 1 # cross-sectional area perpendicular to heat flow
kA = k*area

#boundary terms
bc_type = [1,1]# 1 => essential, 0 => flux
T_0 = 2. # essential BCT at x=0
T_L = 0. # essential BCT at x=1

#forcing function
F = lambda x: 0*x # the forcing function F(x)
alpha = 1.0 # fully implicit
h = np.float(xL - x0)/(n - 1)

C = 4.0
s_cr = 0.5 * C * h**2 / k # critical time step
print s_cr
s = .1 #s_cr

# time domain
t0 = 0
```

```
        tf = 1

        # initial conditions
        IC = np.zeros(len(nodes))

        # array of evaluation times
        t_arr = np.linspace(t0, tf, (tf-t0)/s + 1) # temporal discretization
```

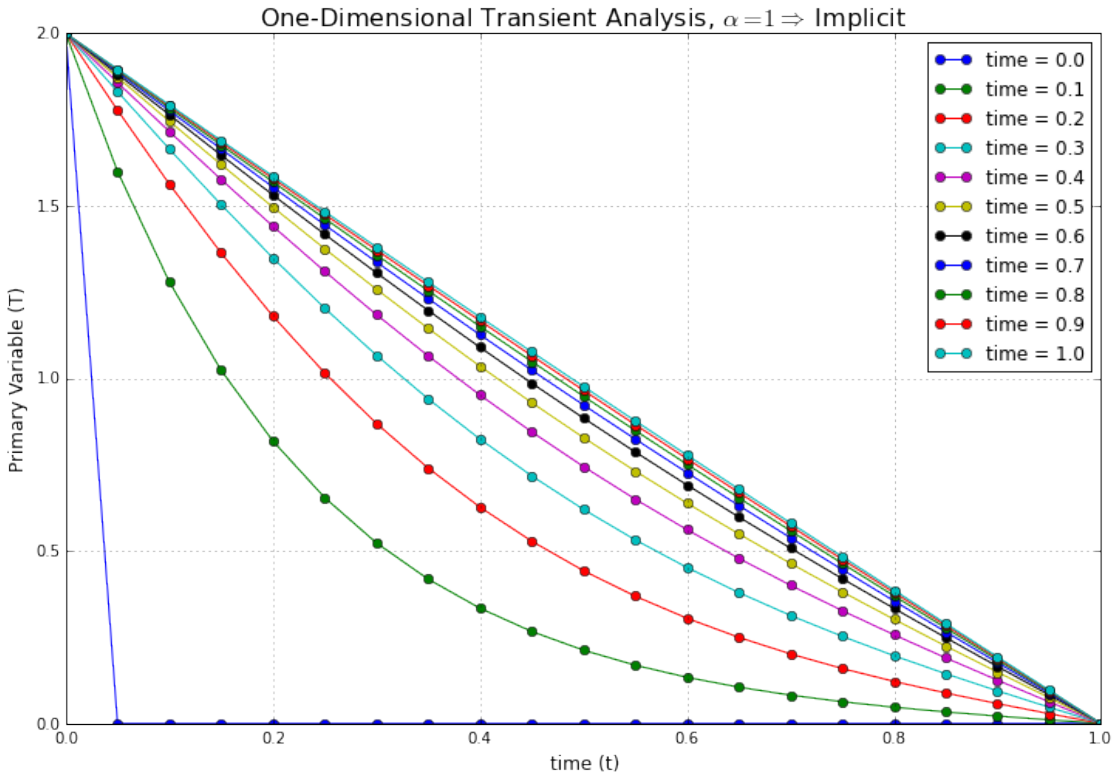
```
0.0025
```

solve problem 4.i

```
In [35]: sol_4i = OneDim_Trans_HeatEqn_FD(nodes,t_arr,IC, C, k, F, bc_type, T_0, T_L, area, alpha) # co
         out_i = sol_4i.solve()
```

Plot results of 4.i

```
In [36]: %matplotlib inline
         import matplotlib.pyplot as plt

         fig_4i, ax = plt.subplots(figsize = (12,8))

         lbl = [None]*len(t_arr)
         for i in xrange(len(t_arr)):
             lbl[i] = 'time = '+ str(t_arr[i])

         # when plotting from arrays, columns from each are plotted against eachother
         ax.plot(nodes, out_i.T,'o-', lw = 1)
         ax.legend(lbl, frameon=1, framealpha = 1, loc=0)

         ax.set_xlabel('time (t)', fontsize = 12)
         ax.set_ylabel('Primary Variable (T)', fontsize = 12)
         ax.set_title(''r'One-Dimensional Transient Analysis, $\alpha = 1 \Rightarrow$ Implicit' , font
         ax.grid(b = True, which = 'major')
         ax.grid(b = True, which = 'major')

         fig_name = '4i.pdf'
         path = '/Users/Lampe/Documents/UNM_Courses/ME-500/HW05/'
         fig_4i.savefig(path + fig_name)
```



Problem 4.ii setup problem 4.ii

In [25]:
```
#define problem discretization
n = 21 # number of nodes
x0 =0 # left boundary
xL = 1 # right boundary
nodes = np.linspace(x0, xL, n)

#constants
k = 2 # thermal conductivity
area = 1 # cross-sectional area perpendicular to heat flow
kA = k*area

#forcing function
F = lambda x: 0*x # the forcing function F(x)
alpha = 1
h = np.float(xL - x0)/(n - 1)

C = 4.0
s_cr = 0.5 * C * h**2 / k # critical time step
print s_cr
s = .1 #s_cr

# time domain
t0 = 0
tf = 1
```

```
        # initial conditions
        IC = np.zeros(len(nodes))

        # array of evaluation times
        t_arr = np.linspace(t0, tf, (tf-t0)/s + 1) # temporal discretization

        #boundary terms
        bc_type = [2,1]# 2 => essential BC func(t),  0 => flux

        T_0 = lambda t: np.sin(10 * t)  # essential BCT at x=0, function of time
        T_L = 0. # essential BCT at x=1
```

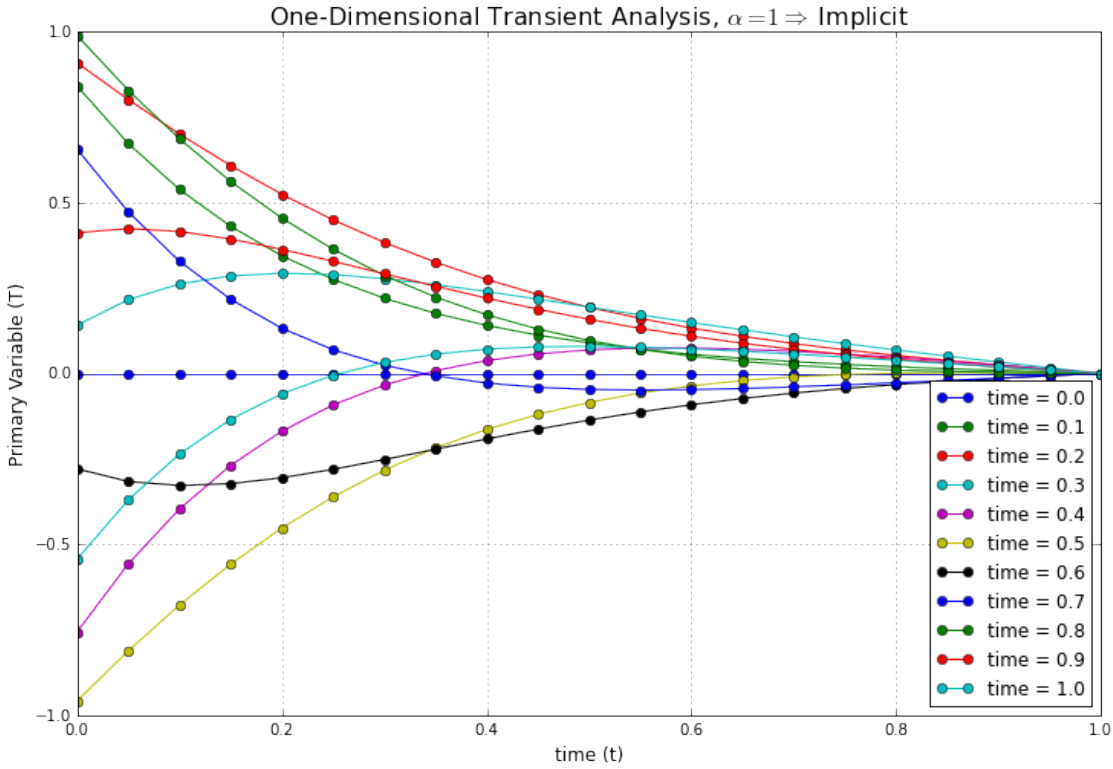
```
0.0025
```

solve 4.ii

```
In [26]: sol_4ii = OneDim_Trans_HeatEqn_FD(nodes,t_arr,IC, C, k, F, bc_type, T_0, T_L, area, alpha) # c
         out_ii = sol_4ii.solve()
```

plot results of 4.ii

```
In [27]: fig_4ii, ax = plt.subplots(figsize = (12,8))

         lbl = [None]*len(t_arr)
         for i in xrange(len(t_arr)):
             lbl[i] = 'time = '+ str(t_arr[i])

         ax.plot(nodes, out_ii.T, 'o-')
         ax.legend(lbl, frameon=1, framealpha = 1, loc=0)

         ax.set_xlabel('time (t)', fontsize = 12)
         ax.set_ylabel('Primary Variable (T)', fontsize = 12)
         ax.set_title(''r'One-Dimensional Transient Analysis, $\alpha = 1 \Rightarrow$ Implicit' , font
         ax.grid(b = True, which = 'major')
         ax.grid(b = True, which = 'major')

         fig_name = '4ii.pdf'
         path = '/Users/Lampe/Documents/UNM_Courses/ME-500/HW05/'
         fig_4ii.savefig(path + fig_name)
```


One-Dimensional Transient Analysis, $\alpha = 1 \Rightarrow$ Implicit
Primary Variable (T)
time (t)
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
time = 0.0
time = 0.1
time = 0.2
time = 0.3
time = 0.4
time = 0.5
time = 0.6
time = 0.7
time = 0.8
time = 0.9
time = 1.0