Finite difference, linear algebra, and tridiagonal matrices

Prepared as part of MOD510 Computational Engineering and Modeling

May 6, 2024

1 Solving the heat equation

Learning objectives:

1. Understand the origin of a conservation equation
2. Formulate a finite difference problem as a matrix inversion problem
3. Quantify the numerical error, investigate the scaling, and analyze the error using Taylors formula
4. Use a sparse solver to speed up the calculation

Exercise 1: Conservation Equation or the Continuity Equation

In figure 1, the continuity equation is derived for heat flow.

Heat equation for solids. As derived in the beginning of this chapter the heat equation for a solid is

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}}{k} = \frac{\rho c_p}{k}\frac{dT}{dt}, \tag{1}$$

where $\dot{\sigma}$ is the rate of heat generation in the solid. This equation can be used as a starting point for many interesting models. In this exercise we will investigate the *steady state* solution, *steady state* is just a fancy way of expressing that we want the solution that *does not change with time*. This is achieved by ignoring the derivative with respect to time in equation (1). We want to study a system with size L, and is it good practice to introduce a dimensionless variable: $y = x/L$.

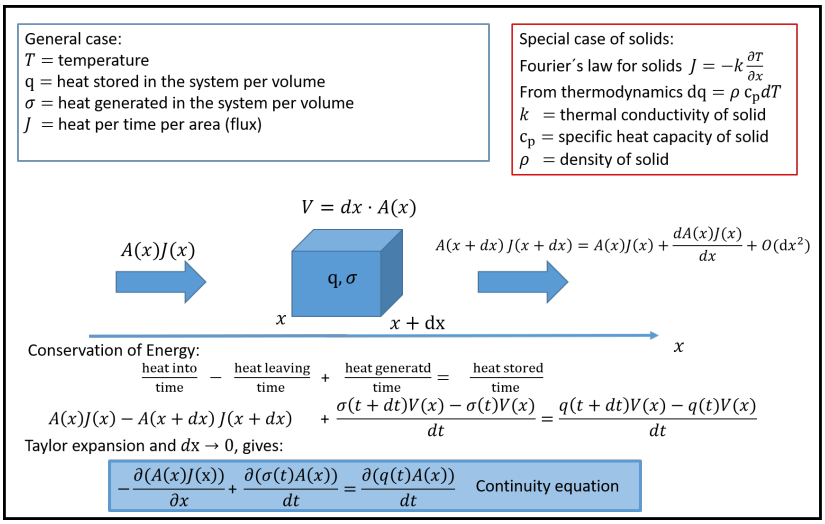

Figure 1: Conservation of energy and the continuity equation.

Part 1. Show that equation (1) now takes the following form:

$$\frac{d^2T}{dy^2} + \frac{\dot{\sigma}L^2}{k} = 0 \tag{2}$$

Solution.

Part 1. Using $d/dx = d/dydy/dx = 1/Ld/dy$, hence $d^2/dx^2 = 1/L^2 d/dy$, and at steady state there are no variation in the heat flow, i.e. $dq/dt = 0$ we arrive at the equation above.

Exercise 2: Curing of Concrete and Matrix Formulation

Curing of concrete is one particular example that we can investigate with equation (2). When concrete is curing, there are a lot of chemical reactions happening, these reactions generate heat. This is a known issue, and if the temperature rises too much compared to the surroundings, the concrete may fracture. In the following we will, for simplicity, assume that the rate of heat generated during curing is constant, $\dot{\sigma}$ =100 W/m^3. The left end (at $x = 0$) is insulated, meaning that there is no flow of heat over that boundary, hence $dT/dx = 0$ at $x = 0$. On the right hand side the temperature is kept constant, $x(L) = y(1) = T_1$, assumed to be equal to the ambient temperature of $T_1 = 25°$C. The concrete thermal conductivity is assumed to be $k = 1.65$ W/m°C.

Part 1. Show that the solution to equation (2) in this case is:

$$T(y) = \frac{\dot{\sigma} L^2}{2k}(1 - y^2) + T_1. \tag{3}$$

Part 2. In order to solve equation (2) numerically, we need to discretize it. Show that equation (2) now takes the following form:

$$T_{i+1} + T_{i-1} - 2T_i = -h^2\beta, \tag{4}$$

where $\beta = \dot{\sigma} L^2/k$.

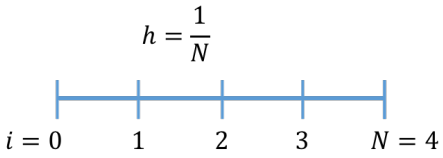

Figure 2: Finite difference grid for $N = 4$.

In figure 2, the finite difference grid is shown for $N = 4$.

Part 3. Show that equation (4) including the boundary conditions for $N = 4$ can be written as the following matrix equation

$$\begin{pmatrix} -\gamma & \gamma & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix}. \tag{5}$$

where $\gamma = 2$ for the central difference scheme and 1 for the forward difference scheme.

Part 4.

- Solve the set of equations in equation (2) using `numpy.linalg.solve`[1].
- Write the code so that you can easily switch between the central difference scheme and forward difference
- Evaluate the numerical error as you change h, how does it scale? Is it what you expect?

```
import numpy as np
import scipy as sc
import scipy.sparse.linalg
from numpy.linalg import solve
import matplotlib.pyplot as plt
```

[1] `https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html`

```
central_difference=False
# set simulation parameters
h=0.25
L=1.0
n = int(round(L/h))
Tb=25 #rhs
sigma=100
k=1.65
beta = sigma*L**2/k

y = np.arange(n+1)*h

def analytical(x):
    return beta*(1-x*x)/2+Tb
def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms
        default k-values for 4x4 matrix:
        | b0 c0 0  0 |
        | a0 b1 c1 0 |
        | 0  a1 b2 c2|
        | 0  0  a2 b3|
    """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)
# defina a, b and c vector
a=np.ones(n-1)
b=..
c=..

if central_difference:
    c[0]= ...
else:
    b[0]=...

A=tri_diag(a,b,c)
print(A) # view matrix - compare with N=4 to make sure no bugs
# define rhs vector
d=...
#rhs boundary condition
d[-1]=...

Tn=np.linalg.solve(A,d)
print(Tn)
```

The correct solution for $L = 1$ m, and $h = 1/4$, is: $[T_0, T_1.T_2, T_3]$=[55.3030303 , 53.40909091, 47.72727273, 38.25757576] (central difference) and $[T_0, T_1.T_2, T_3]$=[62.87878788, 59.09090909, 51.51515152, 40.15151515] (forward difference)

Solution.

Part 1. Integrating equation (2) we get

$$\frac{dT}{dy} = -\beta y + C, \tag{6}$$

where $\beta = \dot{\sigma}L^2/k$. C is an integration constant, using the boundary condition $dT/dy|_{y=0} = 0$, C has to be zero. Integrating the left and right hand side of equation (6) gives us

$$T(y) = -\frac{\beta}{2}y^2 + D. \tag{7}$$

D is a new integration constant, imposing the boundary condition $T(1) = T_1$, we find that $D = T_1 + \beta/2$. Inserting this expression into equation (7), we arrive at equation (3).

Part 2. We replace the second derivative with $dT/dy^2 = (T(y + dy) + T(y - dy) - 2T(y))/dy^2 = (T_{i+1} + T_{i-1} - 2T_i)/h^2$, where we have used the short hand notation $T_i = T(y)$, and replaced dy with h.

Part 3. Let us write down equation (4) for each grid node to see how the implementation is done in practice:

$$\begin{aligned} T_{-1} + T_1 - 2T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + T_4 - 2T_3 &= -h^2\beta. \end{aligned} \tag{8}$$

The tricky part is now to introduce the boundary conditions. The right hand side is easy, because here the temperature is $T_4 = 25$. However, we see that T_{-1} enters and we have no value for this node. The boundary condition on the left hand side is $dT/dy = 0$. We can choose to use the central difference term or the forward difference formulation

$$\left.\frac{dT}{dy}\right|_{y=0} = \frac{T_1 - T_{-1}}{2h} - \frac{h^2}{6}T^{(3)}(\eta) = 0, \tag{9}$$

$$\left.\frac{dT}{dy}\right|_{y=0} = \frac{T_0 - T_{-1}}{h} - \frac{h}{2}T''(\eta) = 0. \tag{10}$$

We have kept the truncation error, hence $T_{-1} = T_1$ for the central difference scheme, and $T_{-1} = T_0$ for the forward difference scheme. Thus the final set of equations are:

$$\begin{aligned} \gamma T_1 - \gamma T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + 25 - 2T_3 &= -h^2\beta, \end{aligned} \tag{11}$$

or in matrix form:

$$\begin{pmatrix} -\gamma & \gamma & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix}. \tag{12}$$

$\gamma = 2$ for the central difference scheme and 1 for the forward difference scheme. Note that it is now easy to increase N as it is only the boundaries that requires special attention.

Part 4.

```
#%%
import numpy as np
import matplotlib.pyplot as plt

central_difference=False
# set simulation parameters
h=0.25
L=1.0
n = int(round(L/h))
Tb=25 #rhs
sigma=100
k=1.65
beta = sigma*L**2/k

y = np.arange(n+1)*h

def analytical(x):
    return beta*(1-x*x)/2+Tb
def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms
        default k-values for 4x4 matrix:
        | b0 c0 0  0 |
        | a0 b1 c1 0 |
        | 0  a1 b2 c2|
        | 0  0  a2 b3|
    """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

a=np.ones(n-1)
b=-2*np.ones(n)
c=np.ones(n-1)
#lhs boundary condition
if central_difference:
    c[0]=2
else:
    b[0]=-1
A=tri_diag(a,b,c)
#rhs vector
```

```
d=np.repeat(-h*h*beta,n)
#rhs boundary condition
d[-1]=d[-1]-Tb
Tn=np.linalg.solve(A,d)

#append boundary temperature
Tn=np.append(Tn,Tb)
#analytical solution
ya=np.arange(0,1,0.01)
Ta=analytical(ya)
#view
if central_difference:
    plt.title('Central Difference Formulation')
else:
    plt.title('Forward Difference Formulation')
plt.plot(ya,Ta,label='analytical')
plt.plot(y,Tn,'*',label='numerical')
plt.legend()
plt.grid()
ni=n//2
yi=y[ni]
err=np.abs(Tn[ni]-analytical(yi))
print('error at point ' + str(ni)+' is='+str(err))
print(Tn)
# %%
```

```
#%%
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.histograms import _hist_bin_auto

def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms
        default k-values for 4x4 matrix:
        | b0 c0 0  0 |
        | a0 b1 c1 0 |
        | 0  a1 b2 c2|
        | 0  0  a2 b3|
    """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

class HeatSolver:
    def __init__(self,h,central=True):
        self.h=h
    # Set simulation parameters
        self.L = 1.0              # length of domain
        self.n = int(round(self.L/self.h))  # number of cells
        self.y=np.arange(self.n+1)*h   # includes right bc
        self.Tb=25
```

```
        self.sigma = 100*self.L**2/1.65
        self.a=np.ones(self.n-1)
        self.b=-np.ones(self.n)*2
        self.c=np.ones(self.n-1)
        self.d=np.repeat(-self.h*self.h*self.sigma,self.n)
        self.d[-1]=self.d[-1]-self.Tb # constant temperature
        #### boundary conditions - central difference ###
        if central:
            self.c[0]=2
        else:
            self.b[0]=-1
        self.A=tri_diag(self.a,self.b,self.c)
    def analytical(self,x):
        return self.sigma*(1-x*x)/2+self.Tb
    def solve(self):
        Tn=np.linalg.solve(self.A,self.d)
        #append boundary value
        self.Tn=np.append(Tn,self.Tb)
    def plot(self):
        ya=np.arange(0,1,0.01)
        Ta=self.analytical(ya)
        plt.plot(self.y,self.Tn,'*', label='numerical n=' + str(self.n))
        plt.plot(ya,Ta, label='analytical' )
        plt.legend()
        plt.grid()
    def error(self):
        return np.abs(self.analytical(0)-self.Tn[0])



H1=HeatSolver(0.025,False)
H1.solve()
H1.plot()
print(H1.error())
h=np.logspace(-4,-1,10)
err=[]
for hi in h:
    H1=HeatSolver(hi,False)
    H1.solve()
    err.append(H1.error())
plt.close()
plt.plot(h,err, '*')
plt.xscale('log')
plt.yscale('log')
plt.grid()
# %%
```

There are two sources to the numerical truncation error, one is from the discretization of the second derivative and one is from the boundary condition. The discretization of the second derivative introduces an error term of the

order $h^4 T^{(4)}$. As we showed earlier the analytical solution is only second order in y, thus the fourth derivative $T^{(4)} = 0$, hence there is *no numerical error from discretization of the second derivative.* The leading order numerical error is introduced at the boundary. If we use the central difference scheme the numerical error is proportional to $h^3 T^{(3)}$ (see equation (10), but this term is also zero as $T^{(3)} = 0$. *Regardless of the number of grid points there are no numerical truncation errors when the central difference scheme is used for the boundary condition.*

For the forward difference scheme the numerical error scales as h and *not* as h^2 as one naively would expect from equation (10) (because $T_{-1} = T_0 + \frac{h^2}{2} T''(\eta)$) thus the first equation in (11) takes the form

$$T_1 - T_0 + \frac{1}{2} h^2 T''(\eta) = -\beta h^2. \tag{13}$$

In this case we can quite easily solve the full system of equations (equation (13) and the three last equations in equation (11))

$$\begin{aligned}
T_0 &= T_1 + 10\beta h^2 + 4\left(\frac{1}{2} h^2 T''(\eta)\right), \\
T_1 &= T_1 + 9\beta h^2 + 3\left(\frac{1}{2} h^2 T''(\eta)\right), \\
T_2 &= T_1 + 7\beta h^2 + 2\left(\frac{1}{2} h^2 T''(\eta)\right), \\
T_3 &= T_1 + 4\beta h^2 + \left(\frac{1}{2} h^2 T''(\eta)\right).
\end{aligned} \tag{14}$$

Notice that the numerical error *accumulates*, if we double the number of grid points the numerical factor in front of the numerical error term in equation (14) doubles. Thus, the number in front of the numerical error *scales as* $1/h$, i.e $1/h = 1/0.25 = 4$. The total numerical error scales as $h^2/h = h$.

Exercise 3: Solve the full heat equation

Part 1. Replace the time derivative in equation (1) with

$$\frac{dT}{dt} \simeq \frac{T(t + \Delta t) - T(t)}{\Delta t} = \frac{T^{n+1} - T^n}{\Delta t}, \tag{15}$$

and show that by using an *implicit formulation* (i.e. that the second derivative with respect to x is to be evaluated at $T(t + \Delta t) \equiv T^{n+1}$) that equation (1) can be written

$$T_{i+1}^{n+1} + T_{i-1}^{n+1} - (2 + \frac{\alpha h^2}{\Delta t}) T_i^{n+1} = -h^2 \beta - \frac{\alpha h^2}{\Delta t} T_i^n, \tag{16}$$

where $\alpha \equiv \rho c_p / k$.

Part 2. Use the central difference formulation for the boundary condition and show that for four nodes we can formulate equation (16) as the following matrix equation

$$\begin{pmatrix} -(2+\frac{\alpha h^2}{\Delta t}) & 2 & 0 & 0 \\ 1 & -(2+\frac{\alpha h^2}{\Delta t}) & 1 & 0 \\ 0 & 1 & -(2+\frac{\alpha h^2}{\Delta t}) & 1 \\ 0 & 0 & 1 & -(2+\frac{\alpha h^2}{\Delta t}) \end{pmatrix} \begin{pmatrix} T_0^{n+1} \\ T_1^{n+1} \\ T_2^{n+1} \\ T_3^{n+1} \end{pmatrix}$$
$$= \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix} - \frac{\alpha h^2}{\Delta t} \begin{pmatrix} T_0^n \\ T_1^n \\ T_2^n \\ T_3^n \end{pmatrix} \tag{17}$$

Part 3. Assume that the initial temperature in the concrete is 25°C, ρ=2400 kg/m^3, a specific heat capacity $c_p = 1000$ W/kg K, and a time step of $\Delta t = 86400$ s (1 day). Solve equation (3), plot the result each day and compare the result after 50 days with the steady state solution in equation (3).

Solution.

Part 1.

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}}{k} = \frac{\rho c_p}{k}\frac{dT}{dt}$$
$$\frac{T_{i+1}^{n+1} + T_{i-1}^{n+1} - 2T_i^{n+1}}{h^2} + \beta = \alpha\frac{T_i^{n+1} - T_i^n}{\Delta t}$$
$$T_{i+1}^{n+1} + T_{i-1}^{n+1} - (2 + \frac{\alpha h^2}{\Delta t})T_i^{n+1} = -h^2\beta - \frac{\alpha h^2}{\Delta t}T_i^n, \tag{18}$$

Part 2. When using the central difference boundary condition, we only need to replace $T_{-1} = T_1$ for $i = 0$, basically follow the same steps as in the previous exercise.

Part 3. Solving the full heat equation, the procedure is as follows

1. choose a value for the step size h, and Δt
2. set up the matrix on the left hand side in equation (3), note that as long as Δt and h is constant it does not change
3. set $T_i^0 = [25, 255, \ldots, 25]$ into the right hand side of equation (3)
4. solve the matrix equation (3) to obtain T_i^1
5. advance clock by Δt (and plot solution)

6. insert T_i^1 in the right hand side of equation (3)
7. solve equation (3) to obtain T_i^2
8. advance clock, and repeat the last two steps

```
#%%
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.histograms import _hist_bin_auto

def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms
        default k-values for 4x4 matrix:
        | b0 c0 0  0 |
        | a0 b1 c1 0 |
        | 0  a1 b2 c2|
        | 0  0  a2 b3|
    """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

class HeatSolver:
    def __init__(self,h):
        self.h=h
        self.dt=60*60*24 # 1 day
        self.t_final=60*60*24*50 # 10 days
    # Set simulation parameters
        self.L = 1.0              # length of domain
        self.k=1.65
        self.cp=1000
        self.rho=2400
        self.n = int(round(self.L/self.h))  # number of cells
        self.y=np.arange(self.n)*h
        self.Tb=25
        self.sigma = 100*self.L**2/1.65
        self.alpha=self.rho*self.cp*self.L*self.L/self.k
        self.a=np.ones(self.n-1)
        self.b=-(2+self.alpha*h*h/self.dt)*np.ones(self.n)
        self.c=np.ones(self.n-1)

        self.T_old=np.repeat(self.Tb,self.n)
        #### boundary conditions - central difference ##
        self.c[0]=2

        self.A=tri_diag(self.a,self.b,self.c)
    def analytical(self,x):
        return self.sigma*(1-x*x)/2+self.Tb
    def solve_time(self):
        self.t=0
        while self.t<self.t_final:
```

```
            self.solve()
            self.T_old=np.copy(self.Tn)
            self.plot()
            self.t +=self.dt

    def solve(self):
        self.d=np.repeat(-self.h*self.h*self.sigma,self.n)
        self.d = self.d-self.alpha*self.h*self.h/self.dt*self.T_old
        self.d[-1]=self.d[-1]-self.Tb # constant temperature
        self.Tn=np.linalg.solve(self.A,self.d)

    def plot(self):
        ya=np.arange(0,1,0.01)
        Ta=self.analytical(ya)
        plt.title('Heat distribution at time ' + str(self.t/(24*60*60))+ ' days')
        plt.plot(self.y,self.Tn,'*', label='numerical n=' + str(self.n))
        plt.plot(ya,Ta, label='analytical' )
        plt.legend()
        plt.grid()
        plt.show()
    def error(self):
        i=self.n//2
        yi=self.y[i]
        return np.abs(self.analytical(yi)-self.Tn[i])



H1=HeatSolver(0.25)
H1.solve_time()

# %%
```

Exercise 4: Using sparse matrices in python

In this part we are going to create a sparse matrix in python and use `scipy.sparse.linalg.spsolve` to solve it. The matrix is created using `scipy.sparse.spdiags`.

Part 1. Extend the code you developed in the last exercises to also be able to use sparse matrices, by e.g. a logical switch. Sparse matrices may be defined as follows

```
import scipy.sparse.linalg

#right hand side
# rhs vector
d=np.repeat(-h*h*beta,n)
#rhs - constant temperature
Tb=25
```

```
d[-1]=d[-1]-Tb
#Set up sparse matrix
diagonals=np.zeros((3,n))
diagonals[0,:]= 1
diagonals[1,:]= -2
diagonals[2,:]= 1
#No flux boundary condition
diagonals[2,1]= 2
A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc')
# to view matrix - do this and check that it is correct!
print(A_sparse.todense())
# solve matrix
Tb = sc.sparse.linalg.spsolve(A_sparse,d)

# if you like you can use timeit to check the efficiency
# %timeit sc.sparse.linalg.spsolve( ... )
```

- Compare the sparse solver with the standard Numpy solver using `%timeit`, how large must the linear system be before an improvement in speed is seen?