Investigation of Numerical Solutions to the Heat Equation

Paul Simmerling, *Electrical Engineering & Physics*, University of Connecticut

***Abstract*—There are many methods for numerically solving partial differential equations, in this paper two of these methods are investigated. First, the Forward Difference method is discussed and used on a test case involving the Heat Equation. The second method was the Crank-Nicolson Method, this method was also discussed and tested on the same Heat Equation. A comparison of difficulty in solving/coding is briefly considered alongside the benefits of each. It is seen that the Crank-Nicolson method is significantly faster (by about a factor of 150x) but has non-physical artifacts.**

I. Introduction

OFTEN in nature, there exist partial differential equations that can not be solved using any know analytic method. However, knowing the behavior of these equations can be extremely important. For example in aerospace, the behavior of how the air flows around the plane is essential for understanding whether or not the plane will be able to fly. The problem arises that the solution to the partial differential equation becomes very complex to the point where there is no analytic solution.

In this report, two numerical methods to solving partial differential equations will be tested alongside their application to the Heat Equation. The first method is the Finite Difference Method where the derivatives will be forward approximated, this will allow us to find an iterative approach to finding an approximate solution to the equation. The Crank-Nicolson Method, will be also investigated where we approximate the derivatives using the central difference approximation. This method will involve the solving systems of linear equations but proves to be much faster.

II. The Heat Equation

The Heat equation is a commonly used partial differential equation in physics and engineering the can easily be thought through qualitatively. This will help us decide whether or not our numerical solutions seem valid without having any experimental data.

For the testing, we will use the one-dimensional heat equation. This means that the equation will only depend on time and one spatial axis. We will choose to model a 1m long aluminum bar that has been heated to 100c. We will then apply the boundary conditions that the ends of the bar are rapidly cooled to 0c. This will allow us to model the cooling of the bar.

$$\frac{\partial T(x,t)}{\partial t} = \frac{\kappa}{C\rho}\frac{\partial^2 T(x,t)}{\partial x^2} \tag{1}$$

Equation 1 is the 1D heat equation. In this equation, we see three physical constants, $\kappa-$ the heat conductivity, $C-$ the specific heat, and ρ - the density of the material. These parameters will determine how quickly the heat flows through the material, for example, is there is a high heat conductivity, the material will cool much faster.

We also see that the equation depends on the first time derivative of the Temperature and the second spatial derivative. This shows us that heat will always flow from the maximum to the minimum temperature as the second derivative of any maximum point is negative.

$$\tilde{T}(x,t) \approx T(x_0,t) + xT'(x_0,t) + x^2\frac{T''(x_0,t)}{2} \tag{2}$$

$$\frac{\partial \tilde{T}(x,t)}{\partial x} = T'(x_0,t) + xT''(x_0,t) \tag{3}$$

$$\frac{\partial^2 \tilde{T}(x,t)}{\partial x} = T''(x_0,t) \tag{4}$$

For a maximum: $T''(x_0,t) < 0$

$$\frac{\partial^2 \tilde{T}(x,t)}{\partial x} < 0 \implies \frac{\partial T(x,t)}{\partial t} < 0 \tag{5}$$

Therefore, heat will flow from the maximum to the minimum. This means in our simulations we

expect the temperature in the bar to rapidly drop near the edges of the bar and be flatter in the center, and will eventually tend towards uniformly 0c.

III. Forward Difference

The first method for numerically solving and simulating the temperature in the bar is the Forward Difference method. For this, we will use the definition of the derivative to approximate the derivatives.

$$\frac{\partial T(x,t)}{\partial t} \cong \frac{T(x,t+\Delta t)-T(x,t)}{\Delta t} \tag{6}$$

$$\frac{\partial^2 T(x,t)}{\partial x^2} \cong \frac{T(x+\Delta x,t)+T(x-\Delta x,t)-2T(x,t)}{(\Delta x)^2} \tag{7}$$

Plugging these back into the heat equation, we can find a discrete form of the equation.

$$\frac{T(x,t+\Delta t)-T(x,t)}{\Delta t} = \frac{\kappa}{C\rho}\frac{T(x+\Delta x,t)+T(x-\Delta x,t)-2T(x,t)}{(\Delta x)^2} \tag{8}$$

Now, we can convert to Latin indices and create the discrete version of the equation using $x = i\Delta x$ and $t = j\Delta t$. We can also simplify our physical constants by creating a new constant, $\mu \equiv \frac{\kappa \Delta t}{C\rho(\Delta x)^2}$.

$$T_{i,j+1} = T_{i,j} + \mu(T_{i+1,j} + T_{i-1,j} - 2T_{i,j}) \tag{9}$$

This lends itself to a simple python code that can solve this to simulate the heat equation.

```
from numpy import *
Nx, Nt = 101, 9000
Dx, Dt = 0.03, 0.9
K, C, P = 237, 900., 2700.
MU = K*Dt/(C*P*Dx**2)
T = zeros((Nx, 2), float)
#Set the entire bar to 100c
T = T+100
#Set the ends of the bar to 0c
T[0,:], T[-1,:] = 0, 0
#Iterate through desired time range
for t in range(1, Nt):
  #Iterate through the bar but
  #don't touch the ends that are 0c
  for ix in range(1, Nx-1):
    T[ix, 1] = T[ix, 0] + MU * \
      ( T[ix+1, 0] + T[ix-1, 0] - \
      2*T[ix,0] )
  #Update T for next iteration
  for ix in range(1, Nx - 1):
    T[ix, 0] = T[ix, 1]
```

Using this code, we can now simulate our aluminum bar. We define how long we want it to run for, $Nt * Dt$, how long the par is $Nx * Dx$, all the physical constants, and initial/boundary conditions. We then iterate through the time we want to simulate and through the length of the bar using the update rule we derived in equation 9. This allowed us to generate the behavior seen in figure 1. Simulating this bar takes approximately 1.5 to 2 seconds.

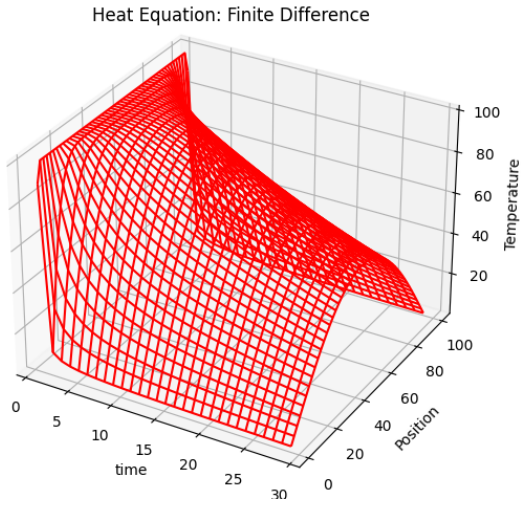

Fig. 1. Simulation of the finite difference method to numerically solve PDEs using the heat equation with an aluminum bar which was initially 100c and edges are fixed at 0c.

From figure 1, we can see that the simulated behavior is quite similar to what we predicted. The edges tend towards 0c much faster than the center of the bar. We also see that the bar is cooling and will eventually reach equilibrium conditions.

IV. Crank-Nicolson

The second numerical method we investigated is the Crank-Nicolson method. Mathematically it is much more involved but has some wonderful advantages over the Forward Difference method.

For this method, we will approximate the derivatives using the central difference approximation for the derivatives.

$$\frac{\partial T(x,t+\frac{\Delta t}{2})}{\partial t} \cong \frac{T(x,t\Delta t)-T(x,t)}{\Delta t} \tag{10}$$

$$\frac{\partial^2 T(x+\frac{\Delta x}{2},t)}{\partial x^2} \cong \frac{T(x-\Delta x,t+\Delta t)-2T(x,t+\Delta t)}{2\Delta x^2} + \frac{T(x+\Delta x,t+\Delta t)}{2\Delta x^2} + \frac{T(x-\Delta x,t)-2T(x,t)+T(x+\Delta x,t)}{2\Delta x^2} \quad (11)$$

We can now plug these back into equation 1 and convert to latin indices to discretize the equation. $x \equiv i\Delta x$, $t = j\Delta t$, and simplifying physical parameter we get $\eta \equiv \frac{\kappa \Delta t}{C\rho \Delta x^2}$.

$$T_{i,j+1} - T_{i,j} = \frac{\eta}{2}[T_{i-1,j+1} - 2T_{i,j+1} + T_{i+1,j+1} + T_{i-1,j} - 2T_{i,j} + T_{i+1,j}] \quad (12)$$

The indices of $j+1$ are future time, $t+\Delta t$, and therefore we need to solve for them. We can rewrite this equation such that these unknown times are on the left and the known T is on the right.

$$-T_{i-1,j+1} + \left(\frac{2}{\eta}+2\right)T_{i,j+1} - T_{i+1,j+1} = T_{i-1,j} + \left(\frac{2}{\eta}+2\right)T_{i,j} + T_{i+1,j+1} \quad (13)$$

This equation can be rewritten in matrix, $\mathbf{A}\vec{x} = \vec{b}$. Note, we know can do some simplification based on the boundary conditions: $T_{0,j+1} = T_{0,j} = T_{N,j+1} = T_{N,j} = 0$.

$$\begin{bmatrix} \left(\frac{2}{\eta}+2\right) & -1 & & \\ -1 & \left(\frac{2}{\eta}+2\right) & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & \left(\frac{2}{\eta}+2\right) \end{bmatrix} \begin{bmatrix} T_{1,j+1} \\ T_{2,j+1} \\ \vdots \\ T_{n-1,j+1} \end{bmatrix} = \begin{bmatrix} T_{0,j+1} + T_{0,j} + \left(\frac{2}{\eta}+2\right)T_{1,j} + T_{2,j} \\ T_{1,j} + \left(\frac{2}{\eta}+2\right)T_{2,j} + T_{3,j} \\ \vdots \\ T_{n-2,j} + \left(\frac{2}{\eta}+2\right)T_{n-1,j} + T_{n,j} + T_{n,j+1} \end{bmatrix} \quad (14)$$

$$\begin{bmatrix} d_1 & c_1 & 0 & 0 \\ a_2 & d_2 & c_2 & 0 \\ 0 & \ddots & \ddots & \ddots \\ 0 & 0 & a_N & d_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} \quad (15)$$

$$\begin{bmatrix} 1 & h_1 & 0 & 0 \\ 0 & 1 & h_2 & 0 \\ 0 & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix} \quad (16)$$

Solving equation 14, we can get it into an upper triangular matrix form (equation 16) where the parameters of $\mathbf{A}$ and $\vec{b}$ are easy to recursively find.

$$h_1 \equiv \frac{c_1}{d_1} \quad (17)$$

$$h_i \equiv \frac{c_i}{d_i - a_i h_{i-1}} \quad (18)$$

$$p_1 \equiv \frac{b_1}{d_1} \quad (19)$$

$$p_i \equiv \frac{b_i - a_i p_{i-1}}{d_i - a_i h_{i-1}} \quad (20)$$

From these values, we can now easy find the values for $\vec{x} = T_{i,j+1}$.

$$x_N = p_N \quad (21)$$

$$x_i = p_i - h_i x_{i-1} \quad (22)$$

We can write a python code to solve this for us.

```
def Tridiag(a,b,c,d,Ta,Tb,Tc,Td,x,n):
  Max = 51
  h, p = zeros((Max),float), \
    zeros((Max),float)
  for i in range(1, n+1):
    a[i], b[i], c[i], d[i] = Ta[i], \
      Tb[i], Tc[i], Td[i]
    #solve for h1 and p1
    h[1], p[1] = c[1]/d[1], b[1]/d[1]
    for i in range(2, n+1):
    #LINE0: Put in equation for
    #solving for h[i]
```

```
    #recursively find h_i from h1
    #  to hn
    h[i] = c[i]/(d[i]-a[i]*h[i-1])
    #LINE1: Put in equation for
    #  solving for p[i]
    #recursively find p_i from p1
    #to pn
    p[i] = (b[i]-a[i]*p[i-1])/ \
        (d[i]-a[i]*h[i-1])
  #we know last row in A has only
  #one value of 1 in col N so
  x[n] = p[n]
  #LINE2: Put in equation for
  #solving for x[i]
  #use backwards substitution
  #from x[n-1] to x[1] knowing x[n]
  #   is p[n]
  #each row has 1 on main diagonal
  #and h_i on upper off diagonal
  for i in range(n-1,0,-1):
    x[i] = p[i] - h[i]*x[i+1]
```

We also need to update the Tb array for each iteration so we are using the correct previous temperature.

```
for j in range(2, m+1):
  print(j)
  for i in range(2, n):
    #reset Tb for next iteration
    Tb[i] = t[i-1,j-1] + \
    t[i+1,j-1] + (2/r - 2) \
      * t[i,j-1]
  #solve matrix
  Tridiag(a,b,c,d,Ta,Tb,Tc,Td,x,n)
  #store temp then into temp now
  for i in range(1, n+1):
    t[i,j] = x[i]
```

Using these algorithms, we can numerically simulate the temperature along the bar. This resulted in the behavior seen in figure 2.

This method simulated the test case in 0.01s to 0.05s. However, there is a strange behavior soon after t=0, the temperature spikes to about -27c. This is not physical and an artifact of our approximations. It does go away and is only a problem at the beginning of the simulation. For most calculations, this shouldn't be a problem but it could cause a butterfly effect if the partial differential equation is not well-posed and exhibits chaotic behaviors.

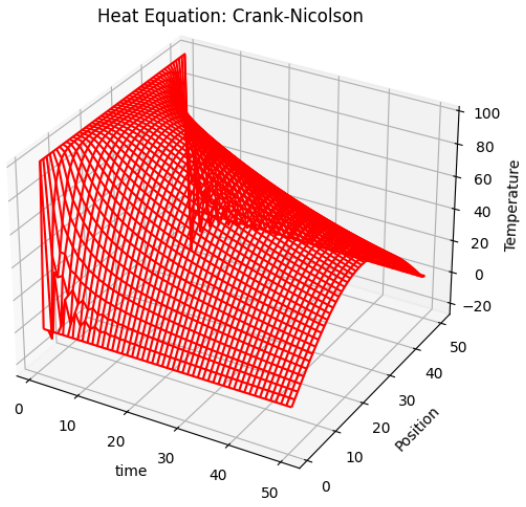

Fig. 2. Simulation of the Crank-Nicolson method to numerically solve PDEs using the heat equation with a bar which was initially 100c and edges are fixed at 0c.

V. Conclusion

Numerically solving is a basis of much of engineering. From exploring space to designing nanoelectronics, simulations and numerical solutions are essential for the designing of complex devices. In this report, we presented two methods to numerically solve partial differential equations. The first method was the Forward Difference method which was mathematically simple and could easily be done. Numerically simulating an aluminum bar with boundary conditions of 0c on the edges of the bar and 100c everywhere else took approximately 1.5s and had no visual defects. Using the Crank-Nicolson approach was more difficult to set up and code but resulted in a simulation that ran significantly faster, 0.01s. However, this method did result in strange non-physical behaviors at the beginning of the simulation. For our test case, it was not an issue but could produce non-realistic results in other simulations.

APPENDIX
NUMERICAL SOLUTION CODE TO THE HEAT EQUATION

```
from math import *
from numpy import *
import time
#Simulation parameters: Range for x, t, and step size for each
Nx, Nt, Dx, Dt = 101, 9000, 0.03, 0.9
#Conductivity, specf heat, density - From measurements on
#  material
KAPPA, SPH, RHO = 237, 900., 2700.
#Solving the PDE using forward differention allows us to simplify
#   and reduce variables
MU = KAPPA / (SPH * RHO) * Dt / ( Dx**2 )
#Make matrix for Temperature -
# use T[:, 1] for time now,
#   T[:, 0] for time 1 dx ago (previous step)
#Tpl is for storing T every k iterations to aid in
#   visualization and graphing
T, Tpl = zeros((Nx, 2), float), zeros((Nx, 31), float)
#LINE0: Insert intial and boundary conditions here
#Set the entire bar to 100c
T = T+100
#Set the ends of the bar to 0c
T[0,:], T[-1,:] = 0, 0

#Iterator for storing T in Tpl
m = 1
#Iterate through desired time range
t0 = time.time()
for t in range(1, Nt):
  #Iterate through the length of the bar but don't touch the ends
  # that are 0c
    for ix in range(1, Nx-1):
        #LINE1: Write update rule here
        #T(x,t+dx) = T(x, t) + MU * ( T(x+dx, t) + T(x-dx, t)
        #   - 2*T(x,t) )
        T[ix, 1] = T[ix, 0] + MU  * ( T[ix+1, 0] + T[ix-1, 0] - \
           2*T[ix,0] )
    #Store T into Tpl so we can plot the time dependence
    if t%300 == 0 or t == 1:
        for ix in range(1, Nx - 1, 2): Tpl[ix, m] = T[ix, 1]
        print(m)
        m = m + 1
    #Update T so T(x,t)=T(x,t+dx) for next iteration
    for ix in range(1, Nx - 1): T[ix, 0] = T[ix, 1]
print("time:",time.time()-t0)

import matplotlib.pyplot as p
from mpl_toolkits.mplot3d import Axes3D
```

```
#Create axes for plots, use every other x
x, y = list(range(1, Nx - 1, 2)), list(range(1, 30))
#Create meshgrid for plotting
X, Y = meshgrid(x, y)

#Find corresponding T values in the mesh grid
def functz(Tpl):
    Z = Tpl[X, Y]
    return Z

Z = functz(Tpl)
#Create figure and plot X Y Z
fig = p.figure()
ax = Axes3D(fig)
ax.plot_wireframe(Y, X, Z, color = 'r')
ax.set_xlabel('time')
ax.set_ylabel('Position')
ax.set_zlabel('Temperature')
fig.suptitle("Heat Equation: Finite Difference")
p.savefig("figures/HE_finite_difference.png")
p.show()


#Create the range for x and t (needs to be symmetric?)
Max, n, m = 51, 50, 50
#Create vector for tridiagonal matrix a is diagonal below main in
#   A, b is standard B,
#c is diag above main diag in A, d  is main diag in A
#T* is for storing and * is for operations
Ta, Tb, Tc, Td = zeros((Max),float), zeros((Max),float), \
  zeros((Max),float), zeros((Max),float)
a,   b,  c,  d = zeros((Max),float), zeros((Max),float), \
  zeros((Max),float), zeros((Max),float)
x, t = zeros((Max),float), zeros((Max,Max),float)

def Tridiag(a, b, c, d, Ta, Tb, Tc, Td, x, n):
    Max = 51
    #Create array for solved upper triangular values h and p
    #h will be primary diag of A, p is the new B
    h, p = zeros((Max),float), zeros((Max),float)
    for i in range(1, n+1):
      #reallocate data
        a[i], b[i], c[i], d[i] = Ta[i], Tb[i], Tc[i], Td[i]
    #solve for h1 and p1
    h[1], p[1] = c[1]/d[1], b[1]/d[1]
    for i in range(2, n+1):
        #LINE0: Put in equation for solving for h[i]
        #recursively find h_i from h1 to hn
        h[i] = c[i] / ( d[i] - a[i] * h[i-1] )
```

```
        #LINE1: Put in equation for solving for p[i]
        #recursively find p_i from p1 to pn
        p[i] = ( b[i] - a[i] * p[i-1] ) / ( d[i] - a[i] * h[i-1] )
    #we know last row in A has only one value of 1 in col N so
    #   solve for Xn=pn
    x[n] = p[n]
    #LINE2: Put in equation for solving for x[i]
    #for i in range(1,n): x[i] = p[i] - h[i]*x[i-1]
    #use backwards substitution fron x[n-1] to x[1] knowing x[n]
    #   is p[n]
    #each row has 1 on main diagonal and hi on main diag+1 =>
    #   x[i]+h[i]*x[i+1]=p[i]
    for i in range(n-1,0,-1): x[i] = p[i] - h[i]*x[i+1]

#create physical device parameters
width, height, ct= 1., .1, 1.
#for i in range(0, n): t[i,0] = 0.
#for i in range(1, m): t[0,i] = 0.
#h = dx, k = dt
h, k = width / ( n - 1 ), height / ( m - 1 )
#r = eta = Conductivity*dt / ( SpecfHeat * density * dx^2 )
#r = KAPPA * k / (SPH * RHO * h**2 )
r = ct**2 * k / ( h**2 )

#Set the edges to 0c
for j in range(1,m+1):
    t[1,j], t[n,j] = 0., 0. #Boundary Condition
#LINE3:t[i][i] = Initial Condition = 100c
for i in range(2,   n): t[i,1]=100
#Set the main diagonal to 2/r + 2 => From Crank-nicolson PDE sln
for i in range(1, n+1): Td[i] = 2. + 2./r
#Make corners 1
Td[1], Td[n] = 1., 1.
#Set the off diagonals = -1
for i in range(1, n): Ta[i], Tc[i] = -1., -1.
#make the off diagonals that don't exist equal to 0, edges BC
Ta[n-1], Tc[1], Tb[1], Tb[n]  = 0., 0., 0., 0.
print("I'm working, wait for fig while I count to 50")

t1 = time.time()
#Iterate through matrix , dont touch BC
for j in range(2, m+1):
    print(j)
    for i in range(2, n):
      #reset Tb for next iteration
        Tb[i] = t[i-1,j-1] + t[i+1,j-1] + (2/r - 2) * t[i,j-1]
    #solve matrix
    Tridiag(a, b, c, d, Ta, Tb, Tc, Td, x, n)
    #store temp then into temp now
    for i in range(1, n+1): t[i,j] = x[i]
```

```
print("Finished")
print("time:",time.time()-t1)
#set up plotting matrix
x, y = list(range(1,m+1)), list(range(1, n+1))
X, Y = meshgrid(x, y)

Z = functz(t)
print(amin(Z))#Getting value below 0! Error with crank-nicolson
#-> does average out though
fig = p.figure()
ax = Axes3D(fig)
ax.plot_wireframe(Y, X, Z, color = 'r')
ax.set_ylabel('Position')
ax.set_xlabel('time')
ax.set_zlabel('Temperature')
fig.suptitle("Heat Equation: Crank-Nicolson")
p.savefig("figures/HE_crank_nicolson.png")
p.show()
```