PHY106 PROJECT

SIMULATING THE CONVECTION-DIFFUSION EQUATION

KAUSTUBH SINGH

SHREYA SAHA

VARSHINI SUBRAMANIAN

SARATH V.S.S

GUHAN KALLAPIRAN

ACKNOWLEDGEMENT

We would like to thank our Professor Priya Johari for giving us the opportunity to perform this project and for continuing guidance throughout the project. We would also like to thank her for giving us an extended deadline to submit our report :)

This project was an amazing learning experience for each one of us and it helped us realize the complexities involved in solving real life problems using computational methods.

ABSTRACT

In this project we aim to compute the Convection-Diffusion equation for which the source information is entered by the user and then proceed to plot a graph depending on the boundary conditions. This can be achieved by solving the equation using finite difference methods. We have then plotted a graph of this equation at various times.

The program has been divided into 2 cases, **1-D system** and **2-D system.**

1-D System

In the one-dimensional case, we are asking the user for the initial and boundary conditions of temperature at time t=0. We are then trying to solve the Convection- Diffusion equation and visually representing the solution using a graph with a slider which we can use to see the system at different time stages.

2-D System

In the two-dimensional case, we again ask for inputs for the initial and boundary conditions and then solve the equation and represent the solution as an animation which shows the temperature at different points as time goes on.

INTRODUCTION:

Convection-diffusion equations are elliptic partial differential equations where the second-order derivatives represent the diffusion, and the first-order derivatives model the convection or transport processes.

Although convection-diffusion equations arise in several situations, the most common source of problems comes from linearization of Navier-Stokes equations, particularly those with large Reynolds number. (Big Reynolds numbers are associated with more turbulent fluid motion; low numbers are characteristic of the much simpler laminar flow.)

The transfer of particles from an area of higher concentration to an area of lower concentration is called *diffusion.* It is governed by the Diffusion Equation:

$$\frac{\partial c}{\partial t} = \nabla.(k\nabla c)$$

The transfer of a substance by bulk movement of molecules is *convection.* It is governed by the convection equation:

$$\frac{\partial c}{\partial t} = -\nabla.(vc)$$

In nature however, the transfer of particles and energy usually happens through a combination of these two processes. So, an equation made by the combination of both the diffusion and convection equations, called the Convection-Diffusion Equation is used.

$$\nabla.(k\nabla c) + \varphi - \nabla.(vc) = \frac{\partial c}{\partial t}$$

THEORY:

The Convection-Diffusion equation, also called the Advection Diffusion equation is a combination of the Diffusion equation and the Convection (Advection) equation.

$$\nabla.(k\nabla c) + \varphi - \nabla.(vc) = \frac{\partial c}{\partial t}$$

$\nabla.(k\nabla c)$ is the Diffusion term and *k* is called diffusivity or the diffusion coefficient.

$\nabla.(vc)$ is the Convection term, *v* is the velocity field and *c* is the function that we solve for.

φ is the source/sink term. This term in entered by the user.

Explicit method :

The explicit method calculates the state of a system at a later time from the state of a system at the current time. In this approach, using a forward difference at time *t* and a second order central difference for the space derivatives:

The explicit method is known to be numerically stable and convergent when $r \leq ½$ which is one of the disadvantages for explicit methods.

Both Neumann boundary conditions and Dirichlet boundary conditions can be used to solve the convection – diffusion equation explicitly.

Implicit method:

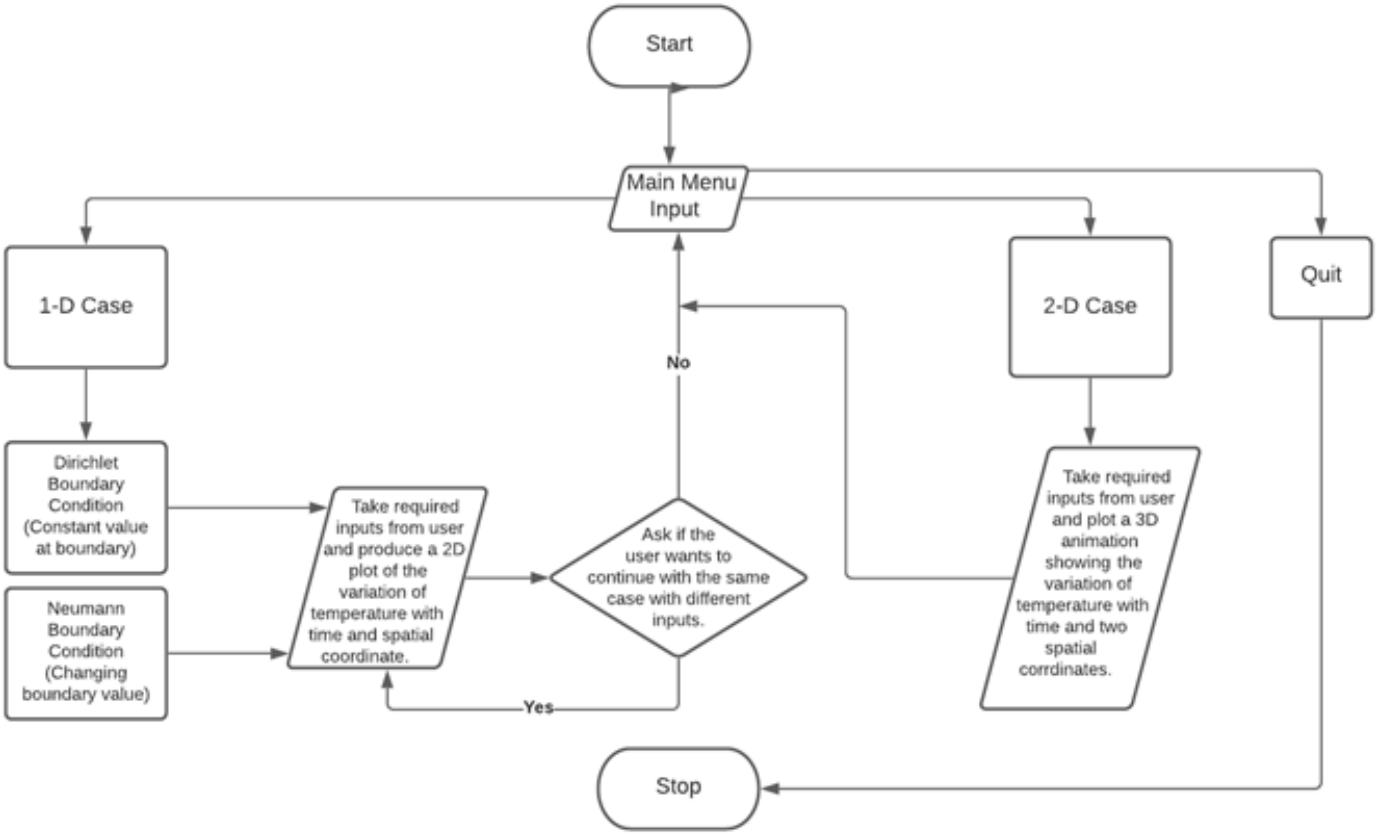
In implicit method, the value of the temperature at a point is dependent on the temperature of some nearby points at the next time step. This results in a system of linear equations which are needed to be solved. The implicit method is preferred because it is stable for any size of time step. Large truncation error due to more procedures might be counted as one of the disadvantages of an implicit method.

Crank Nicolson Method:

In the Crank Nicolson Method, the temperature is equally dependent on t and $t + \Delta t$. $\Delta t \leq h^2/a$ is the only stability critera here, which makes the time step limitation less restricted. The method is based on the finite difference method and is second order accurate in time and space.

In order to get the temperature at the next time step, a tridiagonal system of equations is needed to be solved.

FLOWCHART OF THE PROGRAM



PROGRAM CODE

Explicit method:

```
h_x=1E-1
h_t=1E-2
alpha=1/6

def psi1(x):
    return(np.exp(-(x/2)**2)/8)
def psi2(x):
    return(0)

def v1(x):
    return(x/2)
def v2(z):
    return(2)
def v3(z):
    return(0)
```

Block 1 : Initialising x and t increment while keeping the ratio alpha as a constant equal to 1/6. Alpha is equal to k $\Delta t/(\Delta x)^2$. This ratio is fixed to maintain the stability of the solution.

The source term and the velocity functions are also fixed.

```
def solve(end_t,init_c,xi=-10,xf=10,psi=psi1,v=v2,b_c=[2.3,2.31]):
    '''
    Solves the Diffusion equation numerically by euler's method (Runge- Kutta would be better though)

    end_t : Time till which the solution is computed (starting from 0)
    init_c : Initial Conditions (The value of the temprature at every discreete x value), given in the form of a 1*n array (n is the r
    xi : First x value at which the value of temprature is calculated
    xf : Last x value at which the value of temprature is caluculated
    b_c : Dirichlet boundary conditions, the value at which the temprature is maintained at xi and xf, given in the form of a list

    Returns the value of temprature at every point x domain for every discrete time value in the form of an array,
    who's rows have the value of tempratures at every x value, for a fixed time, and successive rows have tempratures for successive t
    '''
    x_ran=np.arange(xi,xf,h_x)
```

Block 2 : Defining the ‘solve’ function which takes the initial conditions, source term, velocity and boundary condition as input and solves the convection-diffusion equation for the given conditions.

```
x_ran=np.arange(xi,xf,h_x)

#len(init_c) would serve as the number of discrete values of x
ndt=int(end_t/h_t)
#ndt would be the number of discrete values of t, for a small enough h_t, end_t/h_t
y=np.zeros([ndt,len(init_c)])
#a row of y would contain the value of temprature at increasing value of x for cons
y[0]=init_c
#y[0,0]=b_c[0]
#y[0,-1]=b_c[1]
#at time=0 the temprature is given by the initial conditions
for t in range(1,ndt):
    #t takes all the indeces of all the dicreetized time value (except t=0, since w
    for x in range(2,len(init_c)-2):
        #x takes the values of all the indices of the discreetized x values (except
        y[t,x]=alpha*( (y[t-1,x+2] - 2*y[t-1,x] + y[t-1,x-2]) /(4*h_x**2) + psi(x*h
```

```
        #y[t,0]=alpha*(y[t-1,1]-y[t-1,0])*h_t + y[t-1,0]
        y[t,0]=y[t-1,0]
        y[t,1]=y[t,0]
        #y[t,-1]=alpha*(y[t-1,-2]-y[t-1,-1])*h_t + y[t-1,-1
        y[t,-1]=y[t-1,-1]
        y[t,-2]=y[t,-1]
  return(y)
```

Block 3 : Discretizing time and space in the form of arrays within the ‘solve’ function. We are creating ‘y’ array which stores the values of temperature at a given time and space value. t and x takes all the discritized time and space values, except those at the boundaries. y stores the solution of the equation using the forward difference at time t and a second order central difference for the space derivatives.

```
fig, ax = plt.subplots()
plt.grid()
#Initial conditions
k1=1
k2=.5
x_ran=np.arange(-10,10.1,h_x)
#in x_ran, the start and stop values must be the same as those in
s=k1*np.exp(-(x_ran*k2)**2)
s1=np.sin(x_ran*np.pi/(2))
s2=np.sin(x_ran*np.pi)

l, = plt.plot(x_ran, s1, lw=2,color='r')

ax = plt.axis([-10,10,-1,1])
#mind the value of the dimensions here, they must match those in x

axamp = plt.axes([0.25, .04, 0.50, 0.02])
end_time=50
#end time occuers once before in the solve function, independent
samp = Slider(axamp, 'Time', 0, end_time, valinit=0)
y=solve(end_time,s1)
```

```
def update(val):
    # ct is the current value of the slider
    ct = samp.val
    # update curve
    itr=int((ct-.001)/h_t)
    l.set_ydata(y[itr,:])
    # redraw canvas while idle
    fig.canvas.draw_idle()

# call update function on slider value change
samp.on_changed(update)

plt.show()
```

Block 4: The graphing is done in this block S in this block. The arrays s, s1 and s2 define the initial conditions in the equation. The other commands are used for formatting of the graph. The slider is a tool from the library matplotlib which makes it possible to look at the graph of temperature vs x at every time step using the slider. The update function updates the value in the slider at every time step. The values added in y array get graphed at every time step.

Crank Nicolson Method (Implicit Method) : 1D

```
h_x=1E-1
h_t=1E-1
alpha=1/6

def psi1(x):
    return(np.exp(-(x/2)**2)/8)
def psi2(x):
    return(0)


def v1(z):
    return(2)
def v2(z):
    return(0)
```

Block 5 : Initialising x and t increment while keeping the ratio alpha as a constant equal to 1/6. Alpha is equal to k $\Delta t/(\Delta x)^2$. This ratio is fixed to maintain the stability of the solution.

The source term and the velocity functions are also fixed.

```
def solvecn(end_t,init_c,xi=-10,xf=10,psi=psi1,v=v2,b_c=[2.3,2.3]):
    '''

    end_t : Time till which the solution is computed (starting from 0)
    init_c : Initial Conditions (The value of the temprature at every discreete x value), giv
    xi : First x value at which the value of temprature is calculated
    xf : Last x value at which the value of temprature is caluculated
    b_c : Dirichlet boundary conditions, the value at which the temprature is maintained at x

    Returns the value of temprature at every point x domain for every discrete time value in
    who's rows have the value of tempratures at every x value, for a fixed time, and successi
    y[time,position]
    '''
```

Block 6: Defining the ‘solve’ function which takes the initial conditions, source term, velocity and boundary condition as input and solves the convection-diffusion equation for the given conditions.

```
x_ran=np.arange(xi,xf,h_x)
ndx=len(init_c)
#len(init_c) would serve as the number of discrete values of x
ndt=int(end_t/h_t)
#ndt would be the number of discrete values of t, for a small enough h_t, end_t/h_t will
y=np.zeros([ndt,len(init_c)])
#a row of y would contain the value of temprature at increasing value of x for constant t
y[0]=init_c
#y[0,0]=b_c[0]
#y[0,-1]=b_c[1]
#at time=0 the temprature is given by the initial conditions
#Right now, the boundary conditions are what the initial conditions were at the boundarie

A=np.zeros([ndx,ndx])
zeta=h_t/(4*h_x)
xsi=alpha*h_t/(2*h_x**2)

#filling in A
A[0,0]=1
A[-1,-1]=1
```

Block 7 : Discretizing time and space in the form of arrays within the 'solve' function. We are creating 'y' array which stores the values of temperature at a given time and space value. We are also adding the initial conditions to the y array. 'A' is the array which will store a symmetric tridiagonal matrix obtained when we solve the equation using finite difference methods.

```
#Dirichelet boundary conditions
x=xi
for i in range(1,len(A)-1):
    x+=h_x
    A[i,i-1]=-xsi-zeta*v(x-h_x)
    A[i,i]=1+2*xsi
    A[i,i+1]=zeta*v(x+h_x)-xsi

#A[len(A)-1,len(A)-1]=1+2*xsi

c,d,e=tridiag(A)
#The beauty of LU decompostion comes into play here. Since A is not changing (Steady State c
#Therefore we can use the same L and U matrices over and over again, and we only have to so

b=np.zeros([ndx,1])
b[0,0]=y[0,0]
b[-1,0]=y[0,-1]
```

Block 8: The 'A' matrix will not change and the decomposition remains the same for every time step. The 'b' matrix which has been defined is the constant matrix which on the other hand changes in every time step.

```
for t in range(1,ndt):
    x=xi
    for i in range(1,ndx-1):
        x+=h_x
        b[i,0]=psi(x)*h_t/2 + ( (xsi+zeta*v(x-h_x))*y[t-1,i-1] + (1-2*xsi)*y[t-1,i] + (x
    x=sub(b,c,d,e)

    y[t,:]=x.T
return(y,A)
```

Block 9: This block changes the b matrix in every time step according to the finite difference formulas corresponding to the convection diffusion equation and proceeds to solve the matrix equation Ax = b. The solution of this in every time step is added to the ‘y’ array defined before.

```
fig, ax = plt.subplots()
plt.grid()
#Initial conditions
k1=1
k2=.5
x_ran=np.arange(-10,10.1,h_x)
#in x_ran, the start and stop values must be the same as those in the solve function
s=k1*np.exp(-(x_ran*k2)**2)
s1=np.sin(x_ran*np.pi/(2))
s2=np.sin(x_ran*np.pi)

l, = plt.plot(x_ran, s1, lw=2,color='r')

ax = plt.axis([-10,10,-1,1])
#mind the value of the dimensions here, they must match those in x_ran and the values given t

axamp = plt.axes([0.25, .04, 0.50, 0.02])
end_time=50
#end time occuers once before in the solve function, independent of this one
samp = Slider(axamp, 'Time', 0, end_time, valinit=0)

y,A=solvecn(end_time,s1)
def update(val):
    # ct is the current value of the slider
    ct = samp.val
    # update curve
    itr=int((ct-.001)/h_t)
    l.set_ydata(y[itr,:])
    # redraw canvas while idle
    fig.canvas.draw_idle()

# call update function on slider value change
samp.on_changed(update)
plt.show()
```

Block 10: The graphing is done in this block. S arrays here define the initial conditions in the equation. The other commands are used for proper formatting of the graph. The slider is a tool from matplotlib which allows us to look at the graph of temperature vs x at every time step by varying the time using the slider. The update function updates the value in the slider at every time step. The values added previously in the y array are the ones that get graphed at every time step.

This concludes the 1-D case. We now move on to the 2-D case.

2- Dimensional case:

```
xi=-2
xf=2
#square with base from -1 to 1

tf=10
#time till which the simulation would run

hx=1E-1/2
nx= int((xf-xi)/hx)
#number of grid points in x (and the y) axes alone.

ht=1E-2/2
nt=int(tf/ht)

mesh_range = np.arange(xi, xf, hx)
x, y = np.meshgrid(mesh_range, mesh_range,sparse=False)

# Initial condition and the boundary conditions
init_u1 = np.exp(-5 *(x**2 + y**2))
init_u=np.sin(4*x**2+4*y**2)/(x**2+y**2)
bc1=2*mesh_range #left wall
bc2=0*mesh_range #bottom wall
bc3=2*mesh_range #right wall
bc4=0*mesh_range #top wall

alpha=1/5
```

Block 11: We are initialising space and time along with the spatial and time increments for the 2-D case. We are also fixing the initial and boundary conditions.

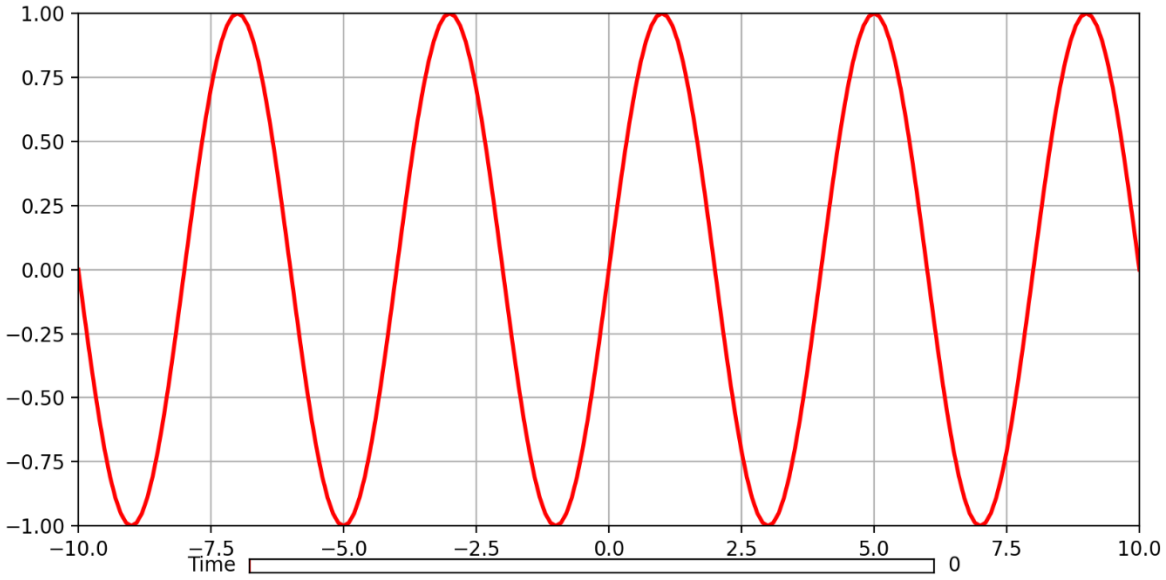
```
def pde_step(u):
    '''
    given the tempratutre distribution u at time t, returns the temprature distribution at t+h
    '''
    #for the time being, the given boundary conditions are not used, the values of the initial
    ut=u.copy()
    for i in range(2,len(u)-2):
        for j in range(2,len(u)-2):
            ut[i,j]=alpha*(u[i+2,j]+u[i-2,j]+u[i,j-2]+u[i,j+2]-4*u[i,j])*ht/(2*hx)**2+u[i,j]
    return(ut)

def draw_plot(x, y, u):
    ax.clear()
    ax.set_zlim(-1.01, 1.01)
    #ax.plot_surface(x, y, U, rstride=1, cstride=1, cmap=cm.coolwarm,
                    #linewidth=0, antialiased=True)
    ax.plot_surface(x,y,u,cmap=cm.rainbow,antialiased=True)
    ax.contourf(x,y,u,cmap=cm.coolwarm,antialiased=True)

    plt.pause(1e-5)
```

Block 12 : The function pde_step uses the explicit scheme to find the temperature at a higher time step given the temperature at a previous time step. The draw_plot function makes a 3d plot for the temperature at different x an y values.

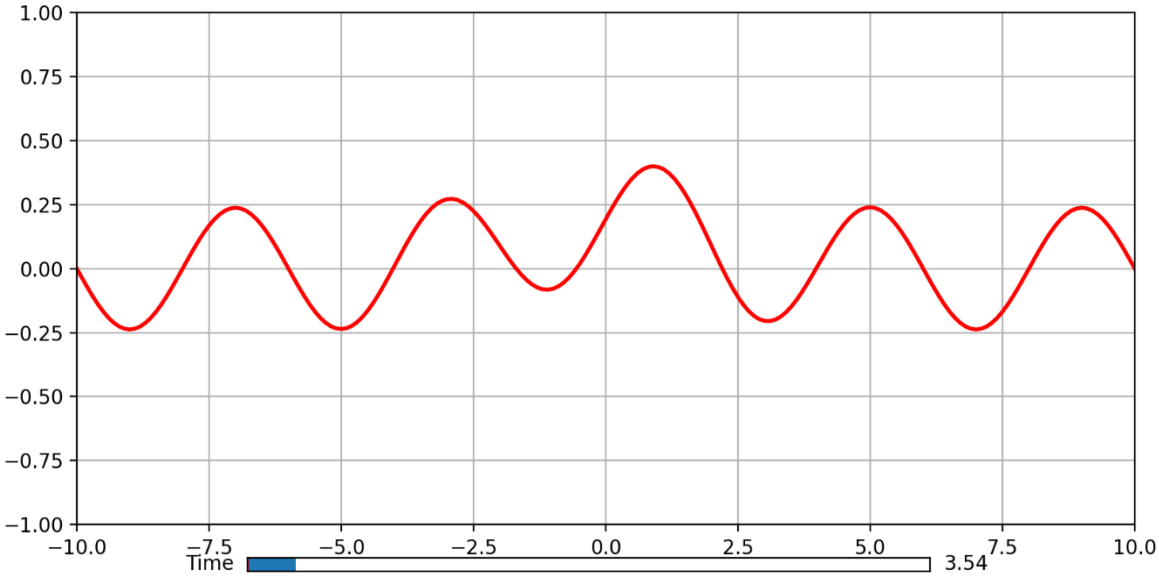
```
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_zlim(-1.01, 1.01)

draw_plot(x, y, init_u)

u_step = init_u
for t in range(nt):
    u_step = pde_step(u_step)

    draw_plot(x, y, u_step)
```

Block 13 : This block uses the previously defined draw_plot function and additional plotting functions to create an animation of the temperature variation with time across the xy plane.

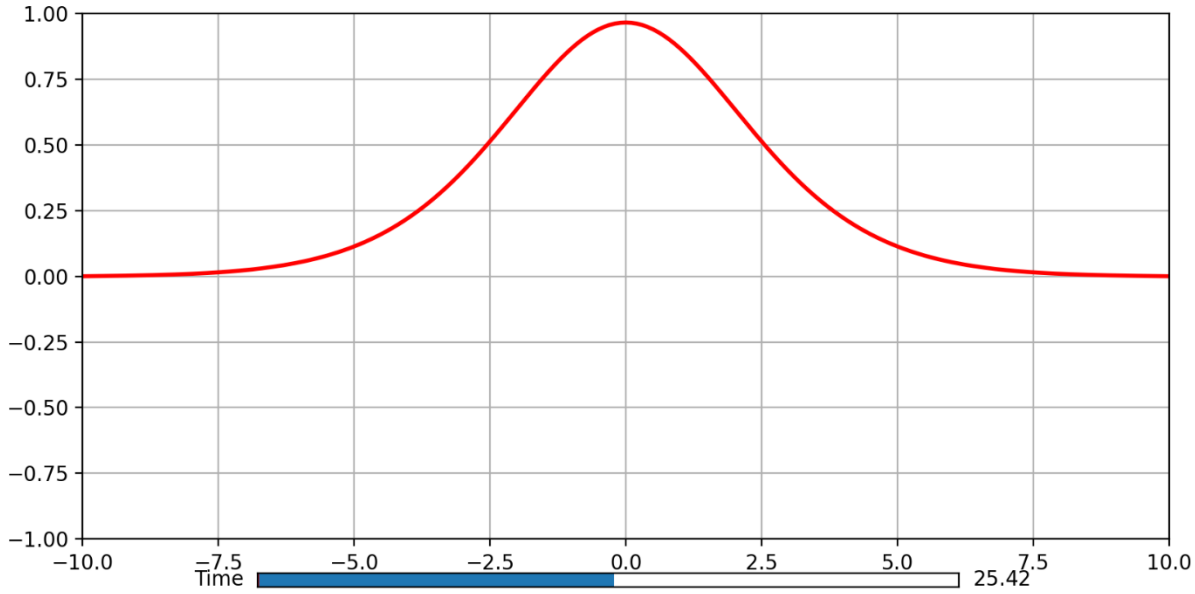
RESULTS OBTAINED FROM THE PROGRAM



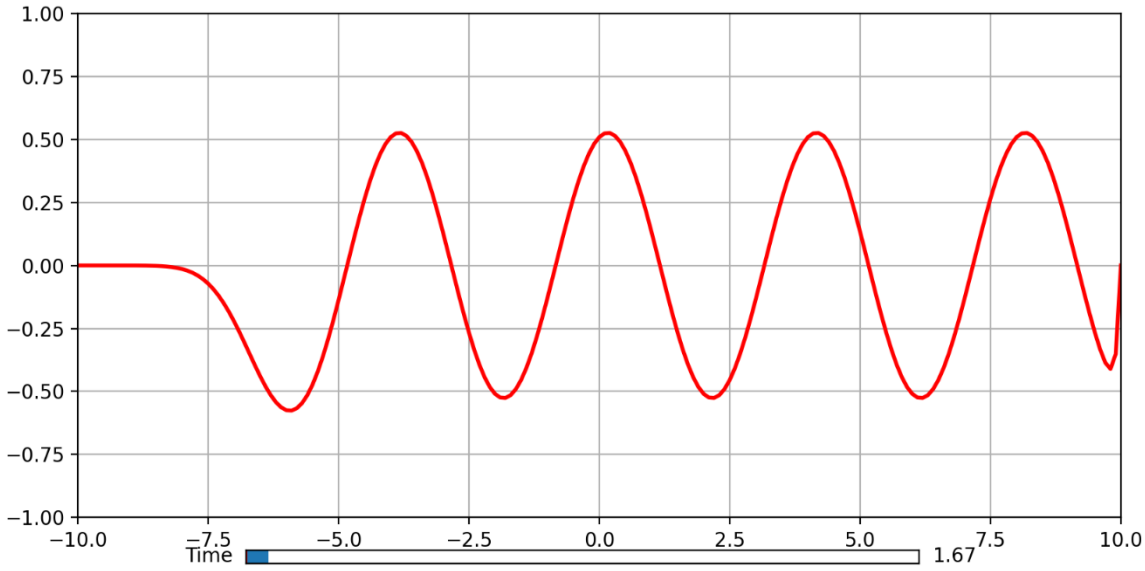
Initial condition of the system



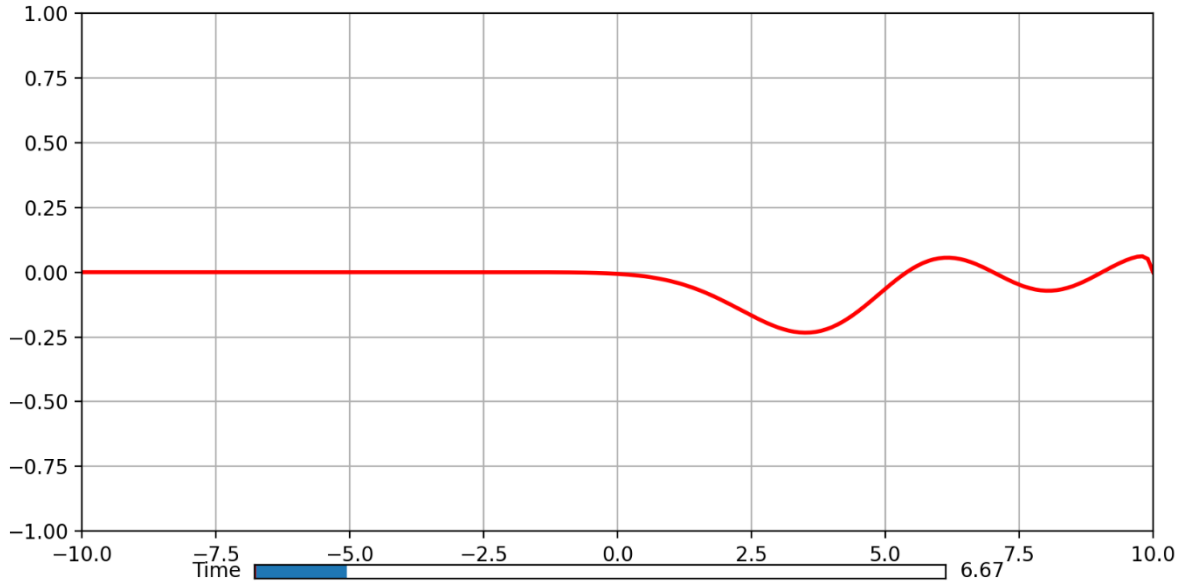
The state of the system after a time of 3.54 seconds. Here, the source term is non-zero and the velocity is zero.



The curve forms this sort of curve as time increases for a non-zero source term and zero velocity.

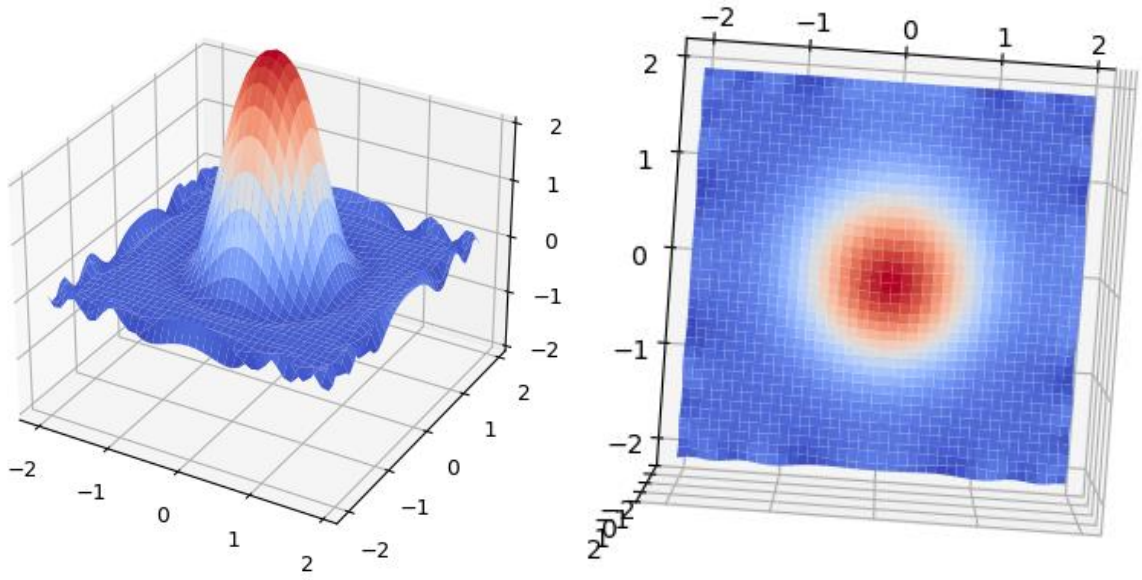


This is the graph for a time t=1.67 seconds for a non-zero velocity and zero source term. We can observe the shift towards the right here caused due to the non-zero velocity term.

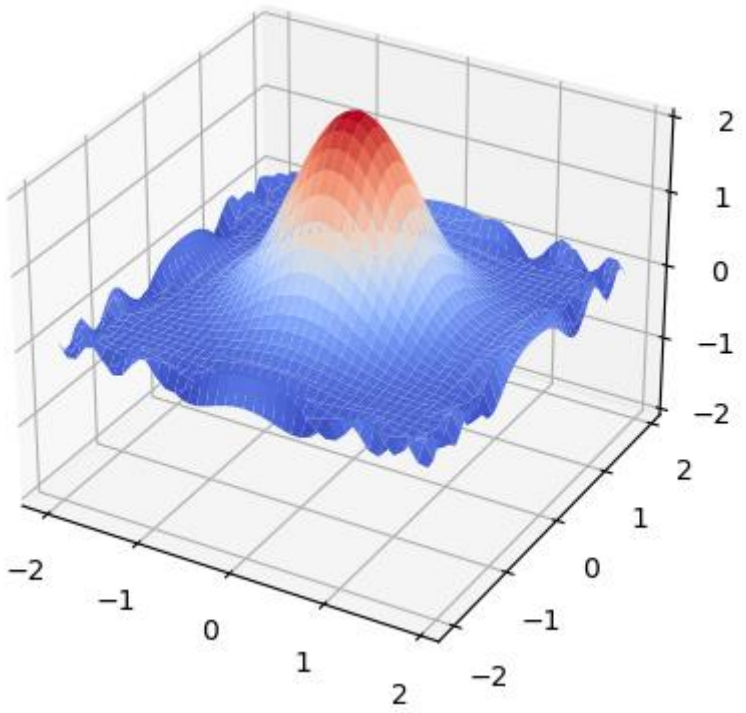


In this case, the graph flattens out as time increases due to the absence of a source term. Eventually it just becomes a flat line.

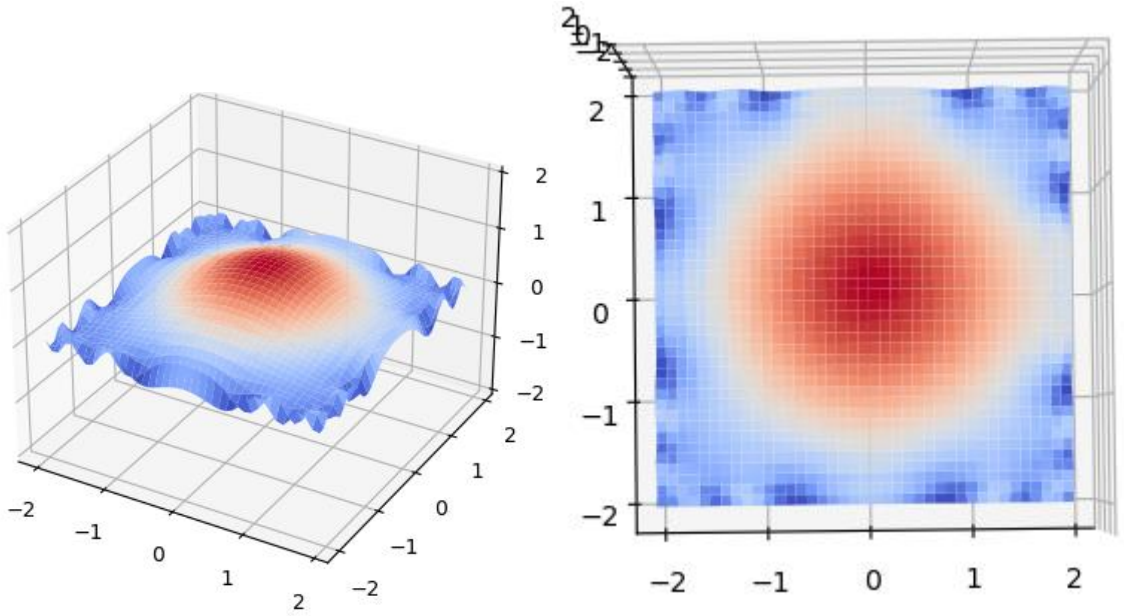
Now for the 2-D case,



This is the initial condition. The difference in colour signifies the variation of temperature over this region.



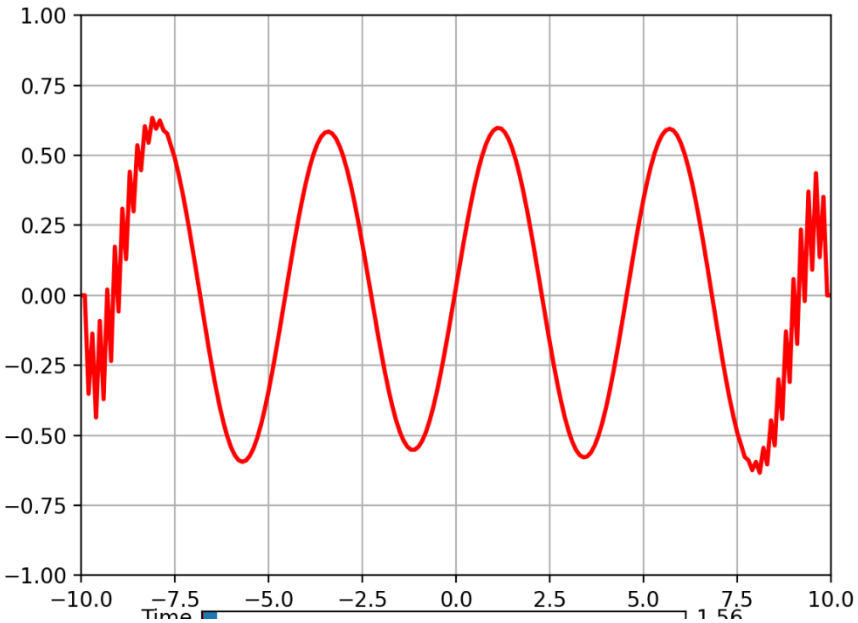
This is the graph after some time has passed. We can see that the curve begins to flatten out.

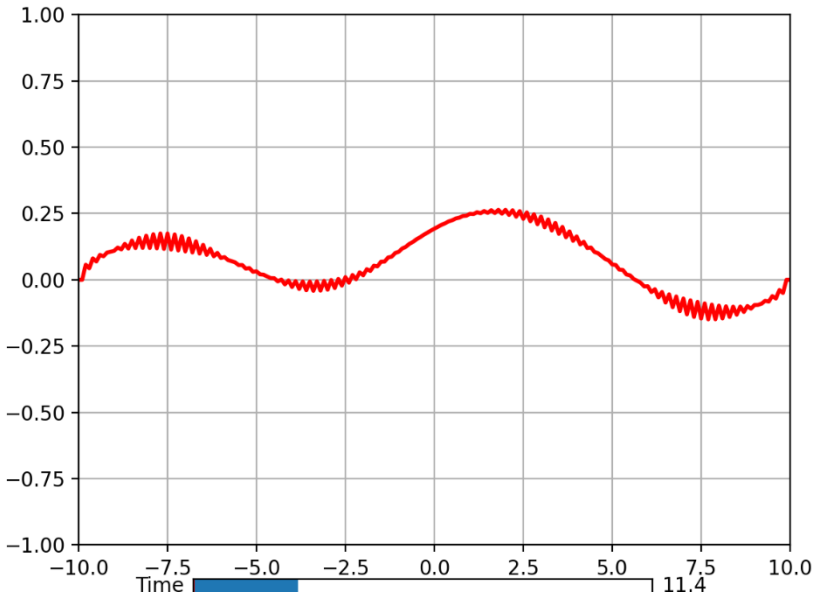


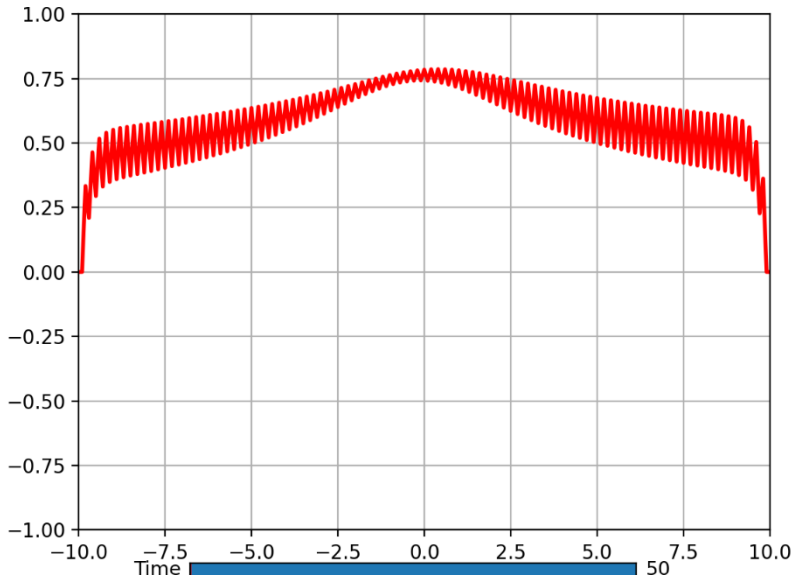
The graph eventually flattens out here due to the absence of a source term.

ERROR ANALYIS:

When using the explicit method for solving the equation, the solutions are not stable for every value of the increment of x and t. The value of alpha (as defined in the program) must be less than ½ for a stable solution. When alpha is equal to 1/6, the error is optimized. When we do not take the proper value of the increments, the graph oscillates, giving skewed results.







As we can see, the errors start from the boundary but eventually creep into the entire curve.

Such errors do not arise in the Implicit methods.

A finite difference method is said to be stable in the naive sense when errors from round off and or truncation do not grow indefinitely.

POSSIBLE IMPROVEMENTS TO THE PROGRAM

- Alternating direction implicit methods (ADI) can be used to solve the equation in the higher dimensional cases.
- The 2-D case can incorporate the Neumann boundary condition in addition to the Dirichlet boundary condition.
- The source and velocity term can be added in the two-dimensional case.
- Modify the 1-D case so that the Neumann boundary conditions with non-zero flux can also be handled.

REFERENCES

- https://www.youtube.com/watch?v=a3V0BJLIo_c&list=PLm3d_4N-H3vwLeDGP5K68G3VgVk0FzLYs&index=5
- https://www.youtube.com/watch?v=rRCGNvMdLEY&list=PLm3d_4N-H3vwLeDGP5K68G3VgVk0FzLYs&index=2
- https://youtu.be/ToIXSwZ1pJU
- https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method
- http://hplgit.github.io/num-methods-for-PDEs/doc/pub/diffu/sphinx/._main_diffu001.html
- https://www.sciencedirect.com/topics/physics-and-astronomy/convection-diffusion-equation
- Numerical solution of the convection–diffusion equation - Wikipedia
- 13717.pdf (uobasrah.edu.iq)