東京工業大学
Tokyo Institute of Technology

PARTIAL DIFFERENTIAL EQUATIONS FOR SCIENCE AND ENGINEERING

FINAL REPORT

AUGUST 5, 2021

Tran Huu Nhat Huy

20B60130

Professor in charge: Dr. Alvin Christopher Varquez

Contents

I. Diffusion Equation

1. Steady state condition

1.1. Methodology

We have the original diffusion equation with constant α:

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0$$

At steady state condition which $\frac{\partial T}{\partial t} = 0$ and under influence of external force $-g$, the original equation turns into:

$$0 - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = -g$$

By using the Forward Time Centered Space (FTCS) method:

$$0 - \alpha\left(\frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{\Delta x^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{\Delta y^2}\right) = -g$$

$$\frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{\Delta x^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{\Delta y^2} = \frac{g}{\alpha}$$

Assuming $\Delta x = \Delta y$ we have:

$$\frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{\Delta x^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{\Delta x^2} = \frac{g}{\alpha}$$

$$T_{i+1,j}^n + T_{i-1,j}^n + T_{i,j+1}^n + T_{i,j-1}^n - 4T_{i,j}^n = \frac{g}{\alpha}\Delta x^2$$

From now, to conduct Poisson numerical modeling for this equation, we use Successive over-relaxation method with algorithm following these steps:

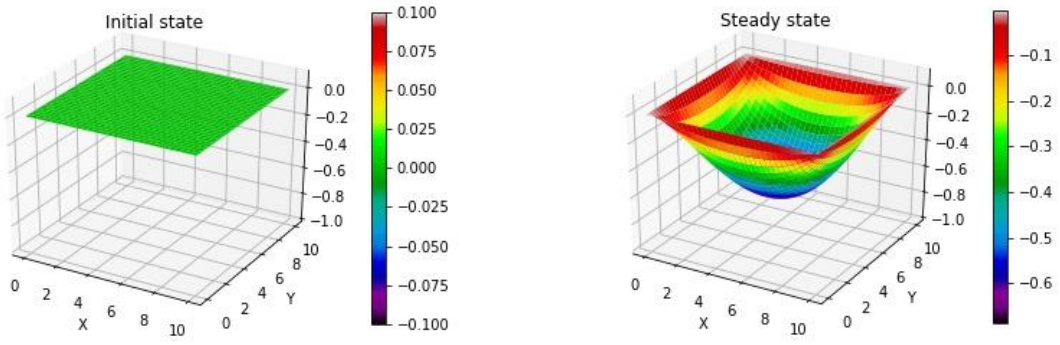
1. Set k number of iterations.
2. Calculate the residual R^k for every iteration.
$$R_{i,j}^k = \frac{1}{4}\left(T_{i+1,j}^k + T_{i-1,j}^{k+1} + T_{i,j+1}^k + T_{i,j-1}^{k+1} - \frac{g}{\alpha}\Delta x^2\right)$$
3. Replace the value of $T_{i,j}^{k+1}$ for the next iteration by adding the residual R^k.
$$T_{i,j}^{k+1} = (1-\omega)T_{i,j}^k + \omega R^k$$

1.2. Source code

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Initial state properties
dx = dy = d = 0.1
alpha = 100
Lx = Ly = L = 10
x = np.arange(0, Lx + dx, dx)
y = np.arange(0, Ly + dy, dy)
g = 9.8

# Simulation runtime properties
X, Y = np.meshgrid(x, y)
T = np.zeros((len(x), len(y)))
fig = plt.figure()

# Function to initiate plot with temperature input T
def init_plt3D(T, zmin = -1, zmax = 0.1):
  ax = fig.add_subplot(projection = '3d')
  plot = ax.plot_surface(X, Y, T, cmap = 'nipy_spectral')
  ax.set_xlabel('X')
  ax.set_ylabel('Y')
  ax.set_zlim(zmin, zmax)
  plt.colorbar(plot)

# Function to simulate steady-state condition (part 1)
def sim_Steady(T):
  T = np.zeros((len(x), len(y)))
  init_plt3D(T)                          # Initial state handling
  plt.title("Initial state")
  plt.savefig("Initial state.jpg")       # Capturing initial state
  w = 1.5                                # Relaxation coefficient (0 < w < 2)
  # Setting Dirichlet boundary condition, fixed at 0
  T[0, :] = T[:, 0] = T[-1, :] = T[:, -1] = 0
  # Commencing simulation
  while (True):
    T_min = T[len(x)//2, len(y)//2]      # Section's center, also lowest point
    for i in range(1, len(x) - 1):
      for j in range(1, len(y) - 1):
        # Calculate residue Rk for every iteration
```

```
        Rk = 1/4 * (T[i+1, j] + T[i-1, j] + T[i, j+1] + T[i, j-
1] - g/alpha*(dx**2))
        # Replace the value of T[i, j] for next iteration by adding Rk
        T[i, j] = (1 - w) * T[i, j] + w * Rk;
    # When the difference of T_min between 2 consecutive iterations is neglig
    ible, we terminate the simulation
    if (abs(T_min - T[len(x)//2, len(y)//2]) <= 0.0001):
      plt.clf()
      init_plt3D(T)
      plt.title("Steady state")
      plt.savefig("steady state.jpg")       # Capturing steady state
      break

# SIMULATION SECTOR
sim_Steady(T)
```

1.3. Result



1.4. Discussion

At first, I decided to loop through a big number of iterations, hoping for the system to reach steady state as close as possible, while capturing every iteration's 3-D graph to make an animation for the simulation. It turned out there were about 1000 images, which made the runtime extremely long. Furthermore, the animation rendering put a greater burden to my CPU and the simulation, as there were . Not only that, this idea of using definite number of iterations seems inefficient, as it is uncertain when the system reaches steady state, thus determining a proper number of iterations is not optimal.

So, I decide to let the iterations continue until they reach a state when the difference between 2 consecutive states is negligible. In particular, the lowest point denoted as T_min is changing at a rate $\Delta z \leq 0.0001$. Then, the simulation will be terminated.

For the result, we can see that from a flat 2-D plane $T(0 \leq x \leq L, 0 \leq y \leq L) = 0$, the system eventually turns into a concave surface, since boundaries are fixed. We can easily see this phenomenon in our normal life, for instance, the observation of a fishing net full of fish being hung. Due to gravity, the fish pull the net down with net's center as lowest point, while net borders are fixed.

2. Other boundary conditions

As in 1.1, we have the original diffusion equation with constant α:

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0$$

Similarly with FTCS method, assuming $\Delta x = \Delta y$:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \alpha\left(\frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{\Delta x^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{\Delta y^2}\right) = 0$$

$$T_i^{n+1} - T_i^n = \frac{\alpha \Delta t}{\Delta x^2}\left(T_{i+1,j}^n + T_{i-1,j}^n + T_{i,j+1}^n + T_{i,j-1}^n - 4T_{i,j}^n\right)$$

Let $d = \frac{\alpha \Delta t}{\Delta x^2}$ and continue:

$$T_i^{n+1} = T_i^n + d\left(T_{i+1,j}^n + T_{i-1,j}^n + T_{i,j+1}^n + T_{i,j-1}^n - 4T_{i,j}^n\right) (*)$$

2.1. Dirichlet boundary condition

2.1.1. Methodology

A system with Dirichlet boundary condition allows heat to exit towards outer environment. For this case, mathematically the temperature T at all boundary points of given section is fixed as:

$$\begin{cases} T(0 \leq x \leq L, 0) = 0 \\ T(0 \leq x \leq L, L) = 0 \\ T(0, 0 \leq y \leq L) = 0 \\ T(L, 0 \leq y \leq L) = 0 \end{cases}$$

These initial conditions will be set as we use equation $(*)$ to conduct simulation for this type of boundary condition.

2.1.2. Source code

These code sections are added to the previous code mentioned above. Below is separated function to obtain *T* under Dirichlet boundary conditions:

```
def get_T_Dirichlet(T, d):
  T[1:-1, 1:-1] = T[1:-1, 1:-1] + d * (T[1:-1, 2:] + T[1:-1, 0:-
  2] + T[2:, 1:-1] + T[0:-2, 1:-1] - 4*T[1:-1, 1:-1])
  return T
```

Function used for simulating 3 cases:

```
def sim_Other_Boundary(T, d):
  T = np.zeros((len(x), len(y)))
  T[40:100, 2:3] = 100
  for i in range(N_iter):
    # Calculate 3 cases
    T_Dirichlet = get_T_Dirichlet(T, d)
    T_Neumann = get_T_Neumann(T, d)
    T_Mixed = get_T_Mixed(T, d)
    # Plot graphs every N_reset iterations
    if (i % N_reset == 0):
      icount = i // N_reset
      init_plt3D(T_Dirichlet, 0, 1)     ; plt.savefig(f"D{icount:05d}.jpg")
      init_plt3D(T_Neumann, 0, 1)       ; plt.savefig(f"N{icount:05d}.jpg")
      init_plt3D(T_Mixed, 0, 1)         ; plt.savefig(f"M{icount:05d}.jpg")
  # Render animations
  !ffmpeg -r 6 -pattern_type glob -i 'D*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p Dirichlet.mp4

  !ffmpeg -r 6 -pattern_type glob -i 'N*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p Neumann.mp4

  !ffmpeg -r 6 -pattern_type glob -i 'M*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p Mixed.mp4
```

For other two functions used to obtain Neumann and Mixed boundary conditions, I will explain it later on.

2.1.3. Result

The animation of the whole process: shorturl.at/gzG28

2.1.4. Discussion

For initial state, I set temperature value of 100 within $T[40 \leq x \leq 100, 2 \leq y \leq 3]$. This is also applied for the other 2 cases. We can clearly see that the heat quickly dissipates all over the region and diffuses to outer environment. Due to this, the total heat of the system decays and gradually approaches 0.

2.2. Neumann boundary condition

2.2.1. Methodology

For this case, the normal derivative of temperature function is fixed, in other word:

$$\frac{\partial T}{\partial x} = G$$

G refers to the boundary condition value, which means a system with Neumann boundary condition has heat exchange rates at boundaries can be expressed with G.

Approximation for the derivative at the boundaries can be obtained by considering an infinitesimal Δx at a boundary, for instance, $(0, j)$:

$$\frac{T_{1,j}^n - T_{-1,j}^n}{2\Delta x} = G \Longrightarrow T_{-1,j}^n = T_{1,j}^n - 2G\Delta x$$

($T_{-1,j}^n$ is a ghost point outside of the $(0 \leq x \leq L, 0 \leq y \leq L)$ boundary)

To obtain expression of $T_{1,j}^n$ we refer to $(*)$ equation with $i = 0$:

$$T_{0,j}^{n+1} = T_{0,j}^n + d\left(T_{1,j}^n + T_{-1,j}^n + T_{0,j+1}^n + T_{0,j-1}^n - 4T_{0,j}^n\right)$$

Substituting obtained $T_{-1,j}^n$ into above expression, we get:

$$T_{0,j}^{n+1} = T_{0,j}^n + d\left(2T_{1,j}^n - 2G\Delta x + T_{0,j+1}^n + T_{0,j-1}^n - 4T_{0,j}^n\right)$$

Similarly with other boundaries:

$$T_{L,j}^{n+1} = T_{L,j}^n + d\left(2T_{L-1,j}^n - 2G\Delta x + T_{L,j+1}^n + T_{L,j-1}^n - 4T_{L,j}^n\right)$$

$$T_{i,0}^{n+1} = T_{i,0}^n + d\left(2T_{i,1}^n - 2G\Delta x + T_{i+1,0}^n + T_{i-1,0}^n - 4T_{i,0}^n\right)$$

$$T_{i,L}^{n+1} = T_{i,L}^n + d\left(2T_{i,L-1}^n - 2G\Delta x + T_{i+1,L}^n + T_{i-1,L}^n - 4T_{i,L}^n\right)$$

To simplify the simulation, we assume $G = 0$ (this means there are no heat exchange at the boundaries, and the section is totally isolated). The four equations for the four boundaries can be rewritten as:

Boundary	Equation
$(0,j)$	$T_{0,j}^n + d\left(2T_{1,j}^n + T_{0,j+1}^n + T_{0,j-1}^n - 4T_{0,j}^n\right)$
(L,j)	$T_{L,j}^n + d\left(2T_{L-1,j}^n + T_{L,j+1}^n + T_{L,j-1}^n - 4T_{L,j}^n\right)$
$(i,0)$	$T_{i,0}^n + d\left(2T_{i,1}^n + T_{i+1,0}^n + T_{i-1,0}^n - 4T_{i,0}^n\right)$
(i,L)	$T_{i,L}^n + d\left(2T_{i,L-1}^n + T_{i+1,L}^n + T_{i-1,L}^n - 4T_{i,L}^n\right)$

These derived equations will be used to conduct simulation for this type of boundary condition later on.

2.2.2. Source code

Below is separated function to obtain T under Neumann boundary conditions:

```
def get_T_Neumann(T, d):
  g = 0
  T[1: -1, 1: -1] += d*(T[1: -1, 2:] + T[1: -1, 0: -2] + T[2:, 1: -
  1] + T[0: -2, 1: -1] - 4*T[1: -1, 1: -1])
  T[:, -1] += d*(T[:, -2] - 4*T[:, -1] + T[:, -
  2] - 2*dx*g + np.roll(T[:, -1], -1, axis = 0) + np.roll(T[:, -
  1], 1, axis = 0))
  T[:, 0] += d*(T[:, 1] - 4*T[:, 0] + T[:, 1] - 2*dx*g + np.roll(T[:, 0],
  -1, axis = 0) + np.roll(T[:, 0], 1, axis = 0))
  T[0, :] += d*(T[1, :] - 4*T[0, :] + T[1, :] - 2*dx*g + np.roll(T[0, :],
  -1, axis = 0) + np.roll(T[0, :], 1, axis = 0))
  T[-1, :] += d*(T[-2, :] - 4*T[-1, :] + T[-2, :] - 2*dx*g + np.roll(T[-
  1, :], -1, axis = 0) + np.roll(T[-1, :], 1, axis = 0))
  # Recalculating corners
  T[:, -1] += d*(T[:, -2] - 4*T[:, -1] + T[:, -
  2] - 2*dx*g + np.roll(T[:, -1], -1, axis = 0) + np.roll(T[:, -
  1], 1, axis = 0))
  T[:, 0] += d*(T[:, 1] - 4*T[:, 0] + T[:, 1] - 2*dx*g + np.roll(T[:, 0],
  -1, axis = 0) + np.roll(T[:, 0], 1, axis = 0))
  T[0, :] += d*(T[1, :] - 4*T[0, :] + T[1, :] - 2*dx*g + np.roll(T[0, :],
  -1, axis = 0) + np.roll(T[0, :], 1, axis = 0))
  T[-1, :] += d*(T[-2, :] - 4*T[-1, :] + T[-2, :] - 2*dx*g + np.roll(T[-
  1, :], -1, axis = 0) + np.roll(T[-1, :], 1, axis = 0))
  return T
```

2.2.3. Result

The animation of the whole process: shorturl.at/fiqH8

2.2.4. Discussion

I change the initial value of temperature from 100 to 1 for the sake of better observation, and the initial region is centralized to $T[20 \leq x \leq 80,30 \leq y \leq 70]$. The value G is also changed from 9.8 in previous cases to 0 to fit the initial conditions mentioned. We can see that, as $G = 0$ there are no heat exchange at the boundaries, and the section is totally isolated. The simulation showed this clearly.

2.3. Mixed boundaries

2.3.1. Methodology

In case of both Dirichlet and Neumann boundary conditions are presented at initial state, we set the initial state as below:

$$\begin{cases} T(0, 0 \leq y \leq L) = 0 \\ T(L, 0 \leq y \leq L) = 0 \\ \dfrac{\partial T}{\partial y_{(y=0)}} = G = 0 \\ \dfrac{\partial T}{\partial y_{(y=L)}} = G = 0 \end{cases}$$

This means that the boundaries along x-axis follow Dirichlet condition, and boundaries along y-axis follow Neumann condition. Heat exits along x-axis's boundaries, but not for y-axis boundaries.

The opposite scenario can also be simulated by adjusting initial conditions in the source code. For the sake of simplification, we will only analyze the above state.

2.3.2. Source code

Below is separated function to obtain T under mixed boundary conditions (Neumann at x borders, Dirichlet at y borders)

```
def get_T_Mixed(T, d):
  g = 0
  T[1: -1, 1: -1] = T[1: -1, 1: -1] + d * (T[1: -1, 2:] + T[1: -1, 0: -
  2] + T[2:, 1: -1] + T[0: -2, 1: -1] - 4*T[1: -1, 1: -1])
  # Let loose the y borders
  T[0, :] += d*(T[1, :] - 4*T[0, :] + T[1, :] - 2*dx*g + np.roll(T[0, :],
  -1, axis = 0) + np.roll(T[0, :], 1, axis = 0))
  T[-1, :] += d*(T[-2, :] - 4*T[-1, :] + T[-2, :] - 2*dx*g + np.roll(T[-
  1, :], -1, axis = 0) + np.roll(T[-1, :], 1, axis = 0))
  T[0, :] += d*(T[1, :] - 4*T[0, :] + T[1, :] - 2*dx*g + np.roll(T[0, :],
  -1, axis = 0) + np.roll(T[0, :], 1, axis = 0))
  T[-1, :] += d*(T[-2, :] - 4*T[-1, :] + T[-2, :] - 2*dx*g + np.roll(T[-
  1, :], -1, axis = 0) + np.roll(T[-1, :], 1, axis = 0))
  return T
```

2.3.3. Result

The animation of the whole process: shorturl.at/izFP9

2.3.4. Discussion

We can see that the heat can freely exit the system through y borders with Dirichlet conditions, but not for x borders with Neumann conditions.

3. Influence of d

3.1. Source code

Below is the function used to conduct simulation and generate animation.

```
def sim_dTest(T):
  #d_list = np.arange(0, 1.05, 0.05)
  d_list = np.arange(0.23, 0.29, 0.01)
  for i in range(len(d_list)):
    T[20:80, 30:70] = 1
  # Capture each state with corresponding d, 100 Dirichlet iterations each
    for j in range(100):
      T = get_T_Dirichlet(T, d_list[i])
      init_plt3D(T, 0, 1)
      plt.title(f"d = {d_list[i]}")
      plt.savefig(f"{i:02d}.jpg")
  # Render animation
  !ffmpeg -r 2 -pattern_type glob -i '*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p dTest.mp4
```

3.2. Result

The animation of $d = [0, 0.05, 0.1, 0.15, 0.2, \dots, 0.95, 1]$: shorturl.at/hv247

The animation of $d = [0.23, 0.24, 0.25, \ldots, 0.29]$: shorturl.at/iuvA3

3.3. Discussion

At first, I have set of values d ranging from 0 to 100, each separated by 0.05. Then, I find Dirichlet behavior of each case. As we can see, the system behaves fine when $d \leq 0.25$, but starts losing stability after $d = 0.3$. Thus, on second test, I limit the range of d to $0.23 \leq d \leq 0.29$, and this time according to the result I can confirm that $d = 0.25$ is the critical threshold for system to be stable. In other word, when $d > 0.25$, the system becomes unstable.

II. Burger's equation

1. Discretization and algebraic equation

We have the original Burger's equation:

$$\frac{\partial u}{\partial t} + a\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) = v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Applying the Forward-in-time and Backward-in-space (FTBS) method for the first derivatives (those in left side):

$$\frac{\partial u}{\partial t} = \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}$$

$$a\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) = a\left(\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + \frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\Delta y}\right)$$

Applying the Centered difference method for the second derivatives in right side:

$$v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = v\left(\frac{u_{i-1,j}^{n} - 2u_{i,j}^{n} + u_{i+1,j}^{n}}{\Delta x^2} + \frac{u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}}{\Delta y^2}\right)$$

Substituting to original equation, assuming $\Delta x = \Delta y$:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + a\frac{2u_{i,j}^{n} - u_{i-1,j}^{n} - u_{i,j-1}^{n}}{\Delta x} = v\frac{u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n} - 4u_{i,j}^{n}}{\Delta x^2}$$

From this, we obtain discretization form of algebraic equation that can later be used in simulation:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \frac{a\Delta t}{\Delta x}\left(2u_{i,j}^{n} - u_{i-1,j}^{n} - u_{i,j-1}^{n}\right) + \frac{v\Delta t}{\Delta x^2}\left(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n} - 4u_{i,j}^{n}\right)$$

2. Differences between linear and non-linear cases for cyclic condition

2.1. Source code

Initial values and functions required for basic operations:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Initial state properties
dx = dy = d = 1
dt = 0.1
v = 1
Lx = Ly = L = 100
x = np.arange(0, Lx + dx, dx)
y = np.arange(0, Ly + dy, dy)

# Simulation runtime properties
N_iter = 10000
N_reset = 100
X, Y = np.meshgrid(x, y)
U = np.zeros((len(x), len(y)))
U[20:80, 20:80] = 1
fig = plt.figure()

# Function to initiate plot with input U
def init_plt3D(U, zmin = 0, zmax = 1):
  plt.clf()
  ax = fig.add_subplot(projection = '3d')
  plot = ax.plot_surface(X, Y, U, cmap = 'nipy_spectral', vmin = zmin, vma
  x = zmax)
  ax.set_xlabel('X')
  ax.set_ylabel('Y')
  ax.set_zlabel('U')
  ax.set_zlim(zmin, zmax)
  plt.colorbar(plot)

# Function to obtain U[n+1] from previous U[n]
def get_U(U, a):
  U = U - a*dt/dx * (2*U - np.roll(U, 1, axis = 0) - np.roll(U, 1, axis =
  1)) + v*dt/(dx**2) * (np.roll(U, 1, axis = 0) + np.roll(U, -
  1, axis = 0) + np.roll(U, 1, axis = 1) + np.roll(U, -1, axis = 1) - 4*U)
  return U
```

Functions used to conduct simulations and render animations:

```
# Function to simulate linear case (when a is constant)
def sim_Linear(U):
  U_lin = U
  a = 0.5
  for i in range(N_iter):
```

```
    U_lin = get_U(U_lin, a)
    # Plot graphs every N_reset iterations
    if (i % N_reset == 0):
      icount = i // N_reset
      init_plt3D(U_lin, 0, 1)
      plt.savefig(f"L{icount:02d}.jpg")
  # Render animation
  !ffmpeg -r 6 -pattern_type glob -i 'L*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p Linear.mp4

# Function to simulate non-linear case (when a is u itself)
def sim_NonLinear(U):
  U_non = U
  for i in range(N_iter):
    a = U_non                  # Update a for every iteration
    U_non = get_U(U_non, a)
    # Plot graphs every N_reset iterations
    if (i % N_reset == 0):
      icount = i // N_reset
      init_plt3D(U_non, 0, 1)
      plt.savefig(f"N{icount:02d}.jpg")
  # Render animation
  !ffmpeg -r 6 -pattern_type glob -i 'N*.jpg' -
  vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -
  pix_fmt yuv420p Nonlinear.mp4

# SIMULATION SECTOR
sim_Linear(U)
sim_NonLinear(U)
```

2.2. Result

The animation obtained from linear case: shorturl.at/nPYZ1

The animation obtained from non-linear case: shorturl.at/dCOUY

2.3. Discussion

$$\begin{cases} dx = dy = d = 1 \\ \Delta t = 0.1 \\ v = 1 \\ U[20 \leq x, y \leq 80] = 1 \\ a = 0.5 \end{cases}$$

The initial state of the system is set to ensure optimal observation as above, while still following the conditions given by Professor.

For linear case, as $a = 0.5$ is fixed, the simulation behaves linearly. As magnitude of U decreases gradually by time and the system approaches toward equilibrium, we see that the crests travel at a constant velocity across the system.

For non-linear case, however, as a is set to U for every iteration, the system tends to reach equilibrium considerably faster than linear case, and it seems that translational velocity of crests decreases over time. Looking back at the equation used for simulation, the velocity factor $\frac{a\Delta t}{\Delta x}$ now becomes $\frac{u\Delta t}{\Delta x}$ and as u decreases over time, we can observe the same behavior with translational velocity of the system.

3. Behavior of u with time in mixed boundary conditions

3.1. Source code

Function to determine close-wall U:

```
# Function to obtain closed-wall condition
def get_U_closedWall(U, a):
  v_adv = a*dt/dx
  v_dif = v*dt/(dx**2)
  Uclone = U - v_adv * (2*U - np.roll(U, 1, axis = 0) - np.roll(U, 1, axis
 = 1)) + v_dif * (np.roll(U, 1, axis = 0) + np.roll(U, -
1, axis = 0) + np.roll(U, 1, axis = 1) + np.roll(U, -1, axis = 1) - 4*U)
  # Recalculating 2 borders
  Uclone[0, :] = U[0, :] - v_adv * (U[0, :] - np.roll(U, 1, axis = 1))[0,
:] + v_dif * (np.roll(U, -
1, axis = 0)[0, :] + np.roll(U, 1, axis = 1)[0, :] + np.roll(U, -
1, axis = 1)[0, :] - 3*U[0, :])
  Uclone[-1, :] = U[-1, :] - v_adv * (U[-1, :] - np.roll(U, 1, axis = 0)[-
1, :] - np.roll(U, 1, axis = 1))[-
1, :] + v_dif * ( np.roll(U, 1, axis = 0)[-
1, :] + np.roll(U, 1, axis = 1)[-1, :] + np.roll(U, -1, axis = 1)[-
1, :] - 3*U[-1, :])
  return Uclone
```

Function to simulate close-wall conditions with linear case:

```
# Function to obtain closed-wall condition
def get_U_closedWall(U, a):
  v_adv = a*dt/dx
  v_dif = v*dt/(dx**2)
```

```
  Uclone = U - v_adv * (2*U - np.roll(U, 1, axis = 0) - np.roll(U, 1, axis
 = 1)) + v_dif * (np.roll(U, 1, axis = 0) + np.roll(U, -
1, axis = 0) + np.roll(U, 1, axis = 1) + np.roll(U, -1, axis = 1) - 4*U)
  # Recalculating 2 borders
  Uclone[0, :] = U[0, :] - v_adv * (U[0, :] - np.roll(U, 1, axis = 1))[0,
:] + v_dif * (np.roll(U, -
1, axis = 0)[0, :] + np.roll(U, 1, axis = 1)[0, :] + np.roll(U, -
1, axis = 1)[0, :] - 3*U[0, :])
  Uclone[-1, :] = U[-1, :] - v_adv * (U[-1, :] - np.roll(U, 1, axis = 0)[-
1, :] - np.roll(U, 1, axis = 1))[-
1, :] + v_dif * ( np.roll(U, 1, axis = 0)[-
1, :] + np.roll(U, 1, axis = 1)[-1, :] + np.roll(U, -1, axis = 1)[-
1, :] - 3*U[-1, :])
  return Uclone
```

3.2. Result

The animation of linear case with mixed borders: shorturl.at/ezANU

3.3. Discussion

Properties of initial state are the same as previous cases. We can see that the heat seems to be able to leak out of the system. Theoretically it is impossible, however, the simulation points out differently. Thus, I suspect there are some problems in the method (I highly suspect the Backward-in-Space one), or in initial values.

III. 1-D Advection equation

We have the original 1-D advection equation:

$$\frac{\partial f}{\partial t} + u\left(\frac{\partial f}{\partial x}\right) = 0$$

Given initial conditions at time $t = 0$, and a cyclic boundary:

$$f(0, x) = \begin{cases} 1\ (40 \leq x \leq 60) \\ 0\ (\text{otherwise}) \end{cases}$$

Given parameters:

- $u = 1.0$
- $\Delta x = 1.0$
- $N_x = 101$

It is stated in Lecture 13 that the linear interpolation achieves stable condition when $0 \leq c \leq 1$, thus I decided to test the performance of each method with $c = 0.5$ as a common standard.

1. Analytical solution

1.1. Methodology

The analytical solution for 1-D advection equation (hyperbolic) is as below:

$$f(t, x) = f(0, x - ut)$$

From this, we get the algebraic form for simulation:

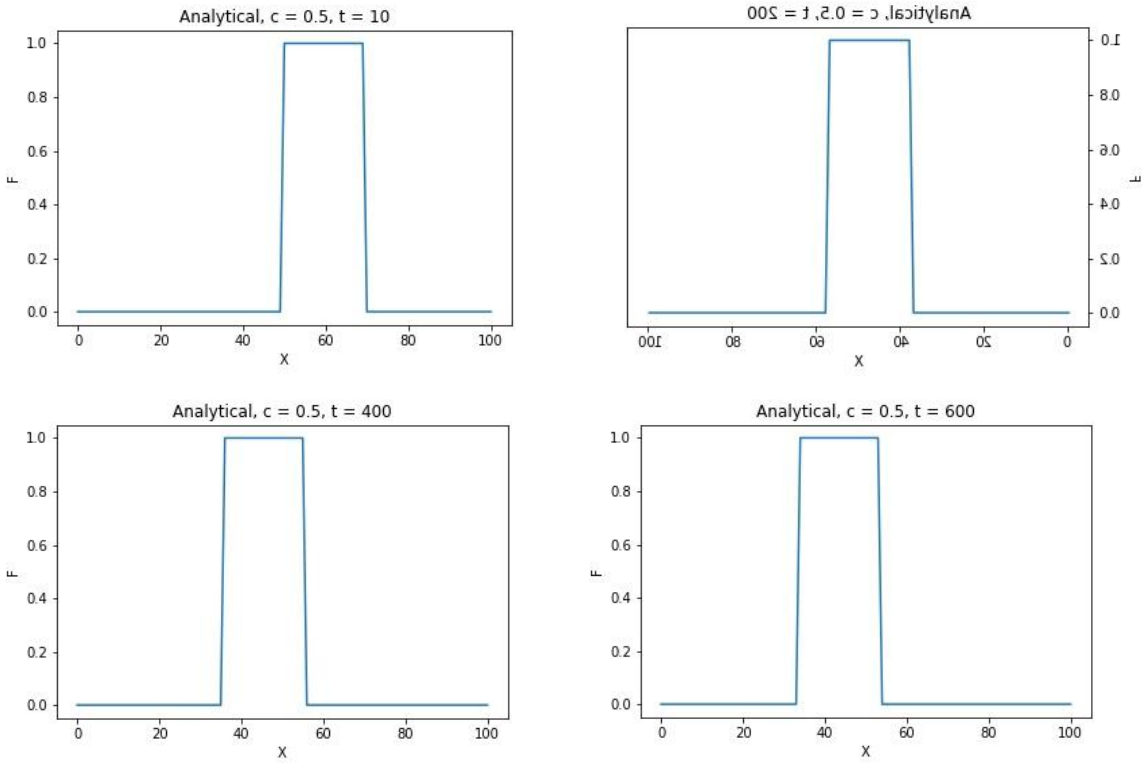
$$f_i^n = f_{i-cn}^0$$

1.2. Source code

Function to conduct simulation:

```
def sim_anal(F, c):
  for t in t_list:
    C = abs(u * t / dx)                          # Courant number
    init_plt2D(np.roll(F, int(C)))               # Plot
    plt.title(f"Analytical, c = {c}, t = {t}")
    plt.savefig(f"A-{c:1.1f}-{t:03d}.jpg")       # Get the graph shot
```

1.3. Result



1.4. Discussion

This is the analytical solution. It seems that the bigger-than-zero region moves toward the left of x-axis as time passes by. We will use this set of graphs to compare and assess other methods later on.

2. Upwind scheme

2.1. Methodology

From the original state, we use Lagrange interpolation (first-order interpolation):

$$\frac{f_i^{n+1} - f_i^n}{u\Delta t} = \frac{f_{iup}^n - f_i^n}{\Delta x} \Longrightarrow f_i^{n+1} = f_i^n + \frac{u\Delta t}{\Delta x}\left(f_{iup}^n - f_i^n\right)$$

The upstream needs to be defined by direction of u. As $u = 1.0 > 0$, we have $iup = i - 1$ and the upstream equation becomes:

$$f_i^{n+1} = f_i^n + \frac{u\Delta t}{\Delta x}(f_{i-1}^n - f_i^n)$$

$$\left(C = \left|\frac{u\Delta t}{\Delta x}\right| \text{ is the Courant number}\right)$$

2.2. Source code

Initial values and settings:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Initial state properties
dx = d = 1
Lx = L = 100
x = np.arange(0, Lx + dx, dx)
u = 1

# Simulation runtime properties
N_iter = 10000
N_reset = 100
F = np.zeros(len(x))
F[40:60] = 1
t_list = [10, 200, 400, 600]
c = 0.5
tMax = max(t_list)
fig = plt.figure()
```

Function to plot graphs:

```
# Function to initiate plot with input function F
def init_plt2D(F):
  plt.clf()
  ax = fig.add_subplot()
  ax.plot(x, F)
  ax.set_xlabel('X')
  ax.set_ylabel('F')
```

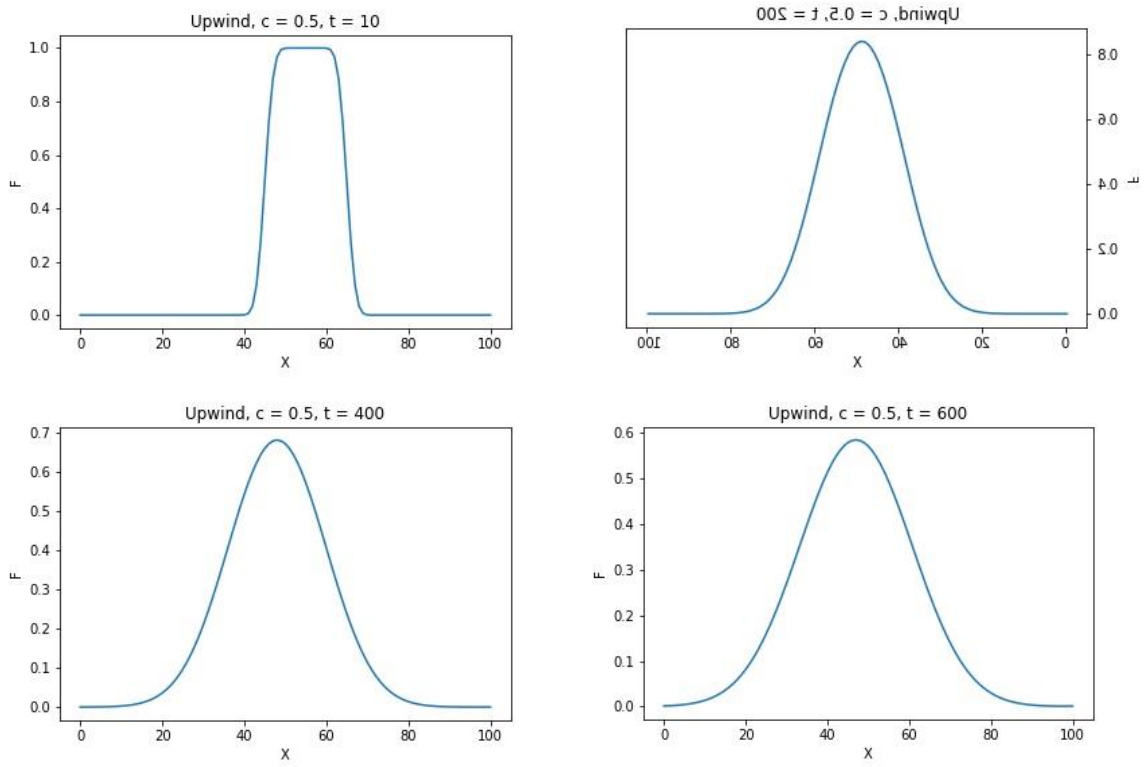
Function to conduct simulation:

```
# Function to simulate up-wind scheme method
def sim_upwind(F):
  F = np.zeros(len(x))
  F[40:60] = 1
  pivotPos = 0
  for t in range(tMax + 1):
    # Approximating F through interpolation loop
    F = F - c * (F - np.roll(F, 1))
    # At each time pivot, plot to see how well F behaves
```

```
if (t == t_list[pivotPos]):
  init_plt2D(F);                              # Plot
  plt.title(f"Upwind, c = {c}, t = {t}")
  plt.savefig(f"U-{c:1.1f}-{t:03d}.jpg")    # Get the graph shot
  pivotPos += 1
```

Other functions will be introduced later on.

2.3. Result



2.4. Discussion

The Upwind scheme method produces a funnel-shaped graph. As time passes, the funnel shape gets bigger and wider.

3. Leith's method

3.1. Methodology

From the original state, we apply second-order interpolation. We know that any second-order polynomial has form $F_i(x) = a_i X_i^2 + b_i X_i + c_i$ with $X_i = (x - x_i)$.

Unknown coefficients:

$$\begin{cases} F_i(x_i) = a_i 0^2 + b_i 0 + c_i = c_i = f_i^n \\ F_i(x_{i+1}) = a_i \Delta x^2 + b_i \Delta x + c_i = f_{i+1}^n \\ F_i(x_{i-1}) = a_i \Delta x^2 - b_i \Delta x + c_i = f_{i-1}^n \end{cases}$$

Substituting the unknown coefficients into second-order polynomial, we get:

$$\begin{cases} c_i = f_i^n \\ b_i = \dfrac{1}{2\Delta x}(f_{i+1}^n - f_{i-1}^n) \\ a_i = \dfrac{1}{2\Delta x^2}(f_{i+1}^n - 2f_i^n + f_{i-1}^n) \end{cases}$$

Then we can get expression of f_i^{n+1} that can be used for simulation:

$$f_i^{n+1} = a_i(u\Delta t)^2 - b_i(u\Delta t) + c_i$$

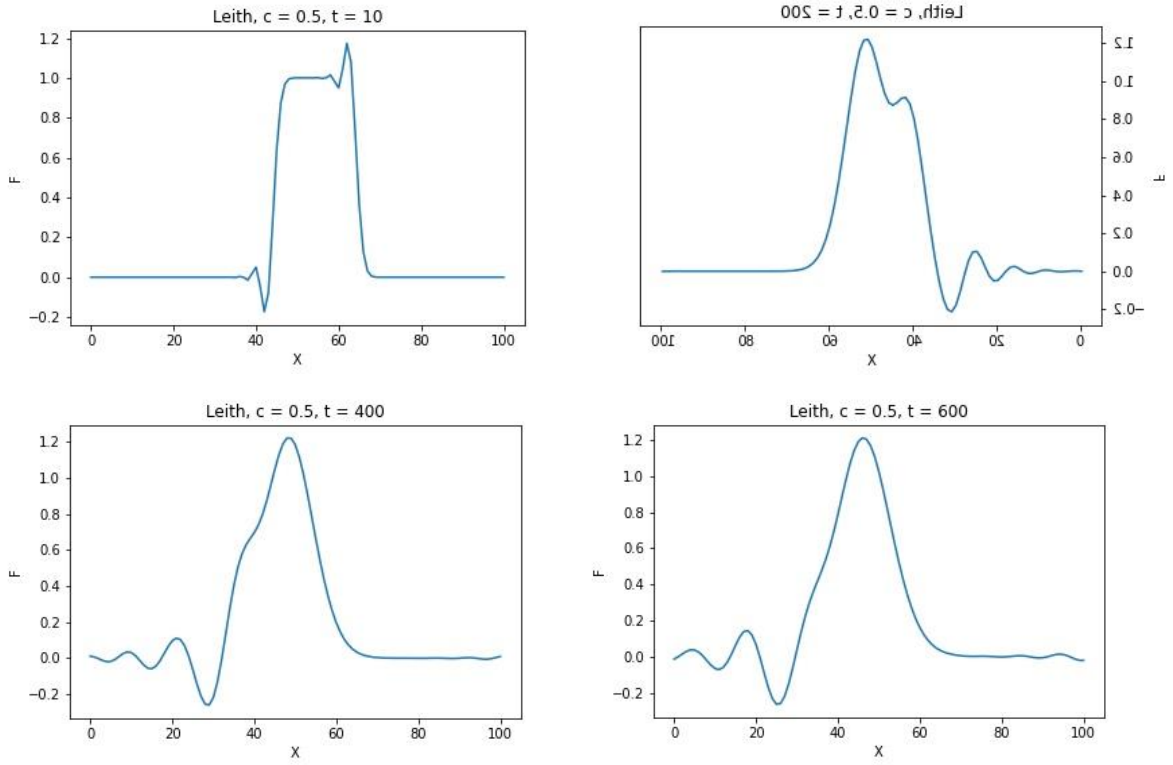
3.2. Source code

Function to conduct simulation:

```
# Function to simulate Leith's method
def sim_Leith(F):
  F = np.zeros(len(x))
  F[40:60] = 1
  pivotPos = 0
  dt = c / u * dx
  for t in range(tMax + 1):
    # Obtain the next interpolation loop
    a = 1 / (2 * (dx**2)) * (np.roll(F, -1) - 2*F + np.roll(F, 1))
    b = 1 / (2 * dx) * (np.roll(F, -1) - np.roll(F, 1))
    F = F + (a * (u*dt)**2) - (b * u*dt)
    # At each time pivot, plot to see how well F behaves
    if (t == t_list[pivotPos]):
      init_plt2D(F);                              # Plot
      plt.title(f"Leith, c = {c}, t = {t}")
      plt.savefig(f"L-{c:1.1f}-{t:03d}.jpg")      # Get the graph shot
      pivotPos += 1
```

3.3. Result

(next page)



3.4. Discussion

We can observe some fluctuations along the graph. As time passes, the graph becomes smoother, yet it seems that the fluctuations prevail.

4. CIP method

4.1. Methodology

From the original state, we apply third-order interpolation. Consider a cubic polynomial with the form $F_i(x) = a_i X_i^3 + b_i X_i^2 + c_i X_i + d_i$

Now we find 4 unknown coefficients with Constrained Interpolated Profile (CIP) method introduced by Professor Takashi Yabe from Tokyo Tech.

2 out of 4 unknown coefficients can be found through:

$$\begin{cases} F_i(x_i) = a_i 0^3 + b_i 0^2 + c_i 0 + d_i = f_i^n \\ F_i(x_{i+1}) = a_i \Delta x^3 + b_i \Delta x^2 + c_i \Delta x + d_i = f_{i+1}^n \end{cases}$$

To find remaining two unknown coefficients, we use derivative with gradient:

$$g_i^n = \left(\frac{\partial f}{\partial x}\right)_i^n$$

The derivative must satisfy g which means:

$$\frac{d}{dx}F_i(x) = 3a_iX_i^2 + 2b_iX_i + c_i$$

To determine g_{i+1}^n we use Hermite interpolation with $\xi = -u\Delta t$:

$$g_{i+1}^n = \frac{d}{dx}F_i(x_p) = 3a_i\xi^2 + 2b_i\xi + g_i^n$$

In the end, we get all 4 unknown coefficients:

$$\begin{cases} a_i = -\dfrac{2(f_{iup}^n - f_i^n)}{\Delta x_{i\to iup}^3} + \dfrac{g_i^n + g_{iup}^n}{\Delta x_{i\to iup}^2} \\ b_i = -\dfrac{3(f_i^n - f_{iup}^n)}{\Delta x_{i\to iup}^2} - \dfrac{2g_i^n + g_{iup}^n}{\Delta x_{i\to iup}} \\ c_i = g_i^n \\ d_i = f_i^n \end{cases}$$

Given that $u = 1.0$ then $iup = i - 1$ and $\Delta x_{i\to iup} = x_{iup} - x_i = \Delta x$ then all the four coefficients can be rewritten as:

$$\begin{cases} a_i = -\dfrac{2(f_{i-1}^n - f_i^n)}{(x_{i-1} - x_i)^3} + \dfrac{g_i^n + g_{i-1}^n}{(x_{i-1} - x_i)^2} \\ b_i = -\dfrac{3(f_i^n - f_{i-1}^n)}{(x_{i-1} - x_i)^2} - \dfrac{2g_i^n + g_{i-1}^n}{x_{i-1} - x_i} \\ c_i = g_i^n = 3a_i\xi_i^2 + 2b_i\xi_i + g_i^{n-1} \\ d_i = f_i^n = a_i\xi_i^3 + b_i\xi_i^2 + g_i^{n-1}\xi_i + f_i^{n-1} \end{cases}$$

Then we can use these equations for simulation later on.

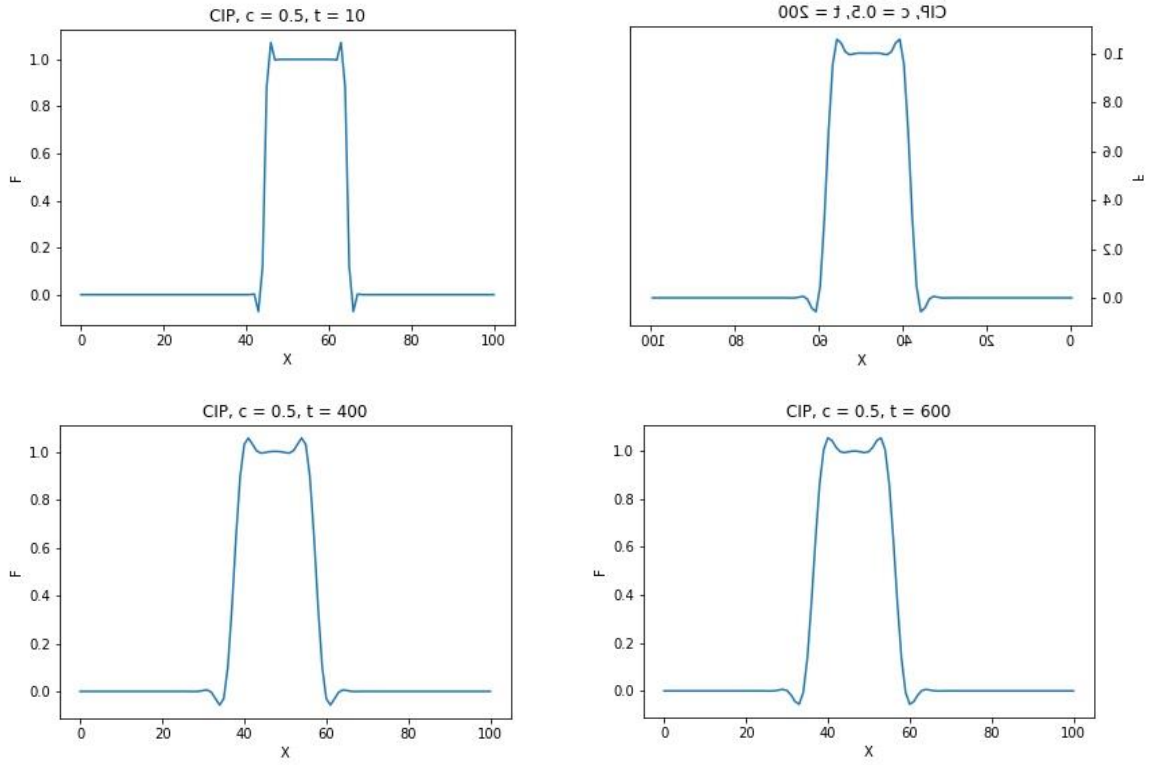
4.2. Source code

Function to conduct simulation:

```
# Function to simulate CIP method
def sim_CIP(F, c):
  F = np.zeros(len(x))
  F[40:60] = 1
  pivotPos = 0
  dt = c / u * dx
  xi = -u * dt
  g = (np.roll(F, -
1) - np.roll(F, 1)) / dx    # g is on tangent line, so g=dy/dx
  for t in range(tMax + 1):
```

```
    # At each time pivot, plot to see how well F behaves
    if (t == t_list[pivotPos]):
      init_plt2D(F);                              # Plot
      plt.title(f"CIP, c = {c}, t = {t}")
      plt.savefig(f"C-{c:1.1f}-{t:03d}.jpg")      # Get the graph shot
      pivotPos += 1
    # Obtain the next interpolation loop
    # We have u = 1 by default, so Xi = dx[i->iup] = x[i-1] - x[i] = -dx
    a = -2 * (np.roll(F, 1) - F) / ((-dx)**3) + (g + np.roll(g, 1)) / ((-
dx)**2)
    b = -3 * (F - np.roll(F, 1)) / ((-dx)**2) - (2*g + np.roll(g, 1)) / (-
dx)
    F = (a * xi**3) + (b * xi**2) + (g * xi) + F
    # Update new value of g for next loop
    g = (3 * a * xi**2) + (2 * b * xi) + g
```

4.3. Result



4.4. Discussion

We obtain a cat-like set of graphs that can be considered as similar to the graph we are trying to imitate. Although there are small fluctuations, we can agree that this result is most similar to analytical one among the three methods.

To sum up, in 3 methods introduced above, the Constrained Interpolated Method, although not totally correct, has the best accuracy when compared with the other two methods. The Upwind scheme method has good accuracy at small t, while the Leith's method has bad accuracy at all values of t.