University of Science and Technology of Hanoi
Department of Space and Application

Practice Report

Climate Modeling

Author:
Nguyen Thi Yen Binh
binhnty.bi12-060@st.usth.edu.vn

Lecturer:
Assoc. Prof. Ngo Duc Thanh
ngo-duc.thanh@usth.edu.vn

December 14, 2023

Contents

List of Figures

1 Introduction

This report presents the solutions and explanations developed for the problems in the "Climate Modeling" course at the University of Science and Technology of Hanoi (USTH), under the guidance of Professor Ngo Duc Thanh.

Climate modeling is the mathematical representations that simulate the various processes shaping the Earth's climate. The objective is to gain quantitative insights into the interactions of physical, chemical, and biological phenomena within the Earth system. This often involves simplifying reality by adjusting physical processes and model resolutions.

The primary goals of climate modeling include developing an understanding of the underlying processes and predicting the consequences of changes within the climate system. However, this endeavor is not without challenges. Climate models, being among the most complex in science, pose computational demands that can be slow and costly or resulting in artificial results that deviating from the physical worlds. As professor Thanh said: "All the simulations are wrong", the models should be used with care which could, in turn, help us understand the physical world better.

Our tools for this practice reports includes the `NumPy` package for efficient array and matrix handling, `pandas` for data manipulation, and `matplotlib` with `scienceplots` for data visualization. All code implementations are carried out in the Jupyter Lab environment within the Anaconda distribution. This report is written in LaTeX.

2 The Atmosphere: vertical distribution

Vertical pressure profile in the atmosphere

$$p(z) = p_0 e^{-z/H}$$

where H=8000(m); p0=1000; Z=0, 1000, ...9000; dz=1000.

1. Plot the vertical profile of pressure
2. Calculate (approximately) the atmospheric density at different levels by using forward, backward, & central difference schemes
3. Compare the obtained simulation results with the analytical results.
4. If dz=500 (i.e. the levels are 0, 500, ..., 9000). Compare the new simulation results with those obtained with dz=1000.
5. Now fix the derivative $dp(z) = -p_0/He^{-z/H}$ à estimate p(z) and compare with the analytical results

2.1 Finite Difference Method

Finite-difference methods (FDM) are a class of numerical techniques for solving differential equations by approximating derivatives with finite differences. Spatial domain and time interval can be broken in a finite number of steps. The value of the solution at these discrete points is approximated by solving algebraic equations containing finite differences and values from nearby points.

This method are widely used for solving ordinary differential equations (ODE) and partial differential equations (PDE), which may be non-linear or their analytical solutions do not exist. Modern computers are used to perform these tasks.

For a n-times differentiable function, by Taylor's theorem the Taylor series expansion is given as

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{f^{(2)}(x)}{2!}(\Delta x)^2 + \frac{f^{(3)}(x)}{3!}(\Delta x)^3 + ... + \frac{f^{(n)}(x)}{n!}(\Delta x)^n + O(x^{n+1})$$

We approximate the Taylor series to first order derivative, lead to 3 basic FDM schemes: Forward, backward and central

Forward Scheme

$$\frac{du}{dt} = \frac{u(x+\Delta x) - u(x)}{\Delta t} + O(x)$$

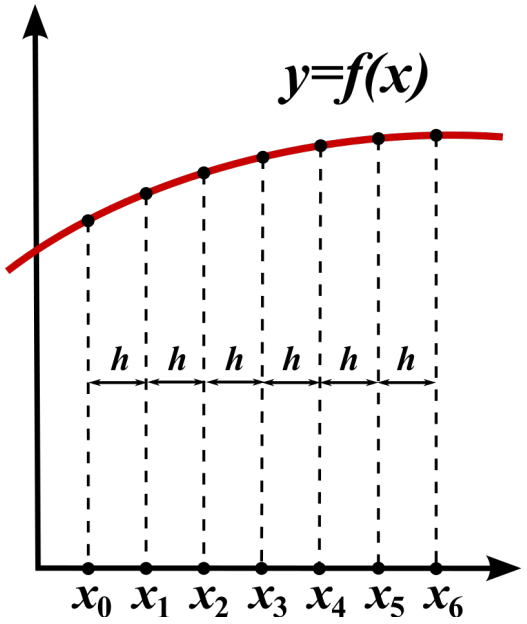


Figure 1: Finite Difference Method

Backward Scheme

$$\frac{du}{dt} = \frac{u(x) - u(x - \Delta x)}{\Delta t} + O(x)$$

Centered Scheme

$$\frac{du}{dt} = \frac{u(x + \Delta x) - u(x - \Delta x)}{2\Delta t} + O(x^2)$$

2.2 Atmospheric Pressure Function

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import scienceplots

plt.style.use(['science','grid'])

# Initial conditions
H = 8000
p0 = 1000
z = np.arange(0,9000,1000)
g = 9.8

# Atmospheric Pressure
```

```
p = lambda z: p0*np.exp(-z/H)

# Plot of atmospheric pressure
plt.figure(figsize=(8,6))
plt.plot(p(z),z)
plt.ylabel("z (m)")
plt.xlabel("p (Pa)")
plt.title("Atmospheric Pressure")

plt.savefig("figures/exercise_figure/ex1_atm_pres.png")
```

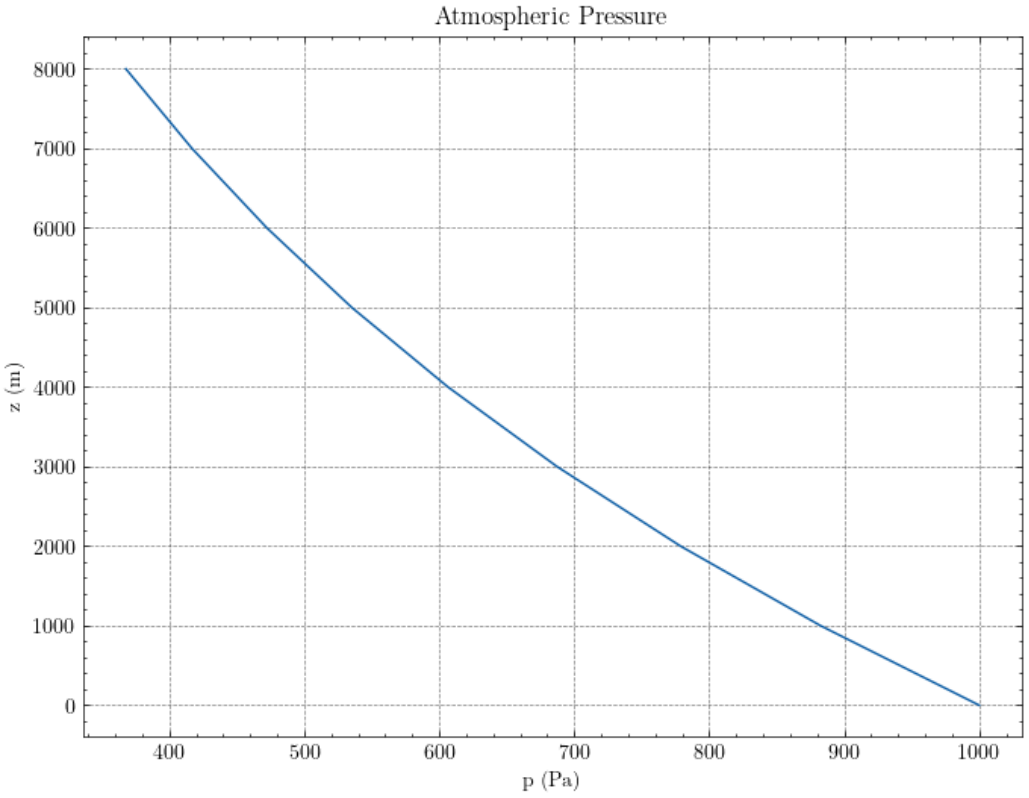


Figure 2: Atmospheric Pressure Diagram with $p_0 = 1000$ Pa

The atmospheric pressure decrease as we go into the higher altitude.

2.3 Atmospheric Density with dz = 1000

```
### 2. Density with forward, backward, central schemeb
dz = z[1] - z[0]
```

```
3
4  # Forward
5  rho_for = -(p(z+dz) - p(z))/(dz*g)
6  # Backward
7  rho_back = -(p(z) - p(z-dz))/(dz*g)
8  # Center
9  rho_central = -(p(z+dz/2) - p(z-dz/2))/(dz*g)
10 # Analytical
11 rho_analytical = 1/(g*H) * p(z)
12
13 # Plot the atmospheric density
14 fig, ax = plt.subplots(1,2,figsize=(16,6))
15
16 ax[0].plot(rho_for,z,label="forward",color='gold')
17 ax[0].plot(rho_back,z,label="backward",color='green')
18 ax[0].plot(rho_central,z,label="center",color='blue')
19 ax[0].plot(rho_analytical,z,linewidth=2,linestyle='--',label='analytical',
       color="black")
20 ax[0].set_xlabel(r'$\rm \rho (g.cm^{-3})$')
21 ax[0].set_ylabel("z (m)")
22 ax[0].set_title("Atmospheric density with dz = 1000")
23 ax[0].legend()
24
25 ax[1].plot(rho_for - rho_analytical, z, label = 'forward', color = 'gold')
26 ax[1].plot(rho_back - rho_analytical, z, label = 'backward', color = 'green'
       )
27 ax[1].plot(rho_central - rho_analytical, z, label = 'center', color = 'blue'
       )
28 ax[1].set_ylabel("z")
29 ax[1].set_xlabel(r"$\rho_{scheme} - \rho_{analytical}$")
30 ax[1].set_title("Accuracy of the schemes for atmospheric density with dz =
       1000")
31 ax[1].legend()
32
33 plt.tight_layout()
34 plt.savefig("figures/exercise_figure/ex1_atm_dens_accur.png")
```

The atmospheric density can be related to the atmospheric pressure by the hydrostatic equation:

$$\frac{dp}{dz} = -\rho g \tag{1}$$

The higher we go, the lower the pressure resulting in lower density atmosphere.

In order to approximate the derivative of pressure p, we use the 3 basic schemes for derivatives in the numerical methods: forward, backward and central introduced in section 2.1.

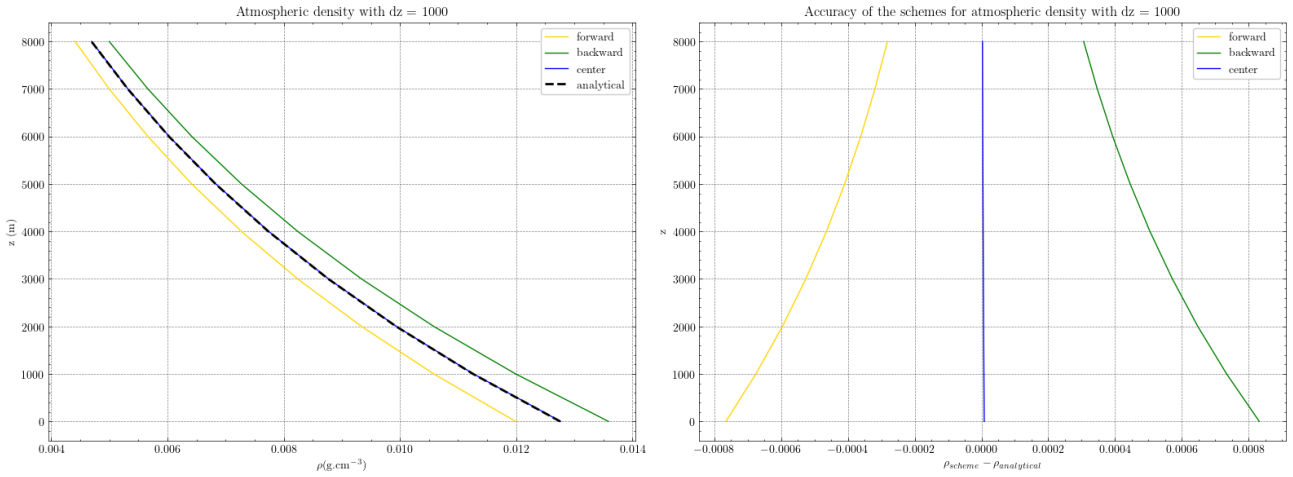


Figure 3: *Left:* The atmospheric density diagram by forward, backward and centered schemes. *Right:* The accuracy of schemes. The closer the value to 0, the more accurate the scheme.

From the figure 3, with dz = 1000, the forward scheme gives the curve with lower value than the analytical curve, while the backward scheme do exactly opposite. From the figure on the right, the accuracy of forward and backward schemes increase as the altitude increase. For the central scheme, the accuracy is very high because the blue line aligns perfectly with the analytical curve, and the accuracy line on the right is nearly zero.

2.4 Atmospheric Density with dz = 500 and dz = 1000

```
dz = 500
rho_for1 = -(p(z+dz) - p(z))/(dz*g)
rho_back1 = -(p(z) - p(z-dz))/(dz*g)
rho_central1 = -(p(z+dz/2) - p(z-dz/2))/(dz*g)
rho_analytical1 = 1/(g*H) * p(z)

fig, ax = plt.subplots(2,3,figsize=(20,10))

# Forward scheme
ax[0,0].plot(rho_for1, z, label="dz = 500")
ax[0,0].plot(rho_for, z, label="dz = 1000")
ax[0,0].plot(rho_analytical, z, color="red",linestyle="--",label='analytical
    ')
ax[0,0].set_title("Forward Scheme")
ax[0,0].set_xlabel(r'$\rm \rho (g.cm^{-3})$')
ax[0,0].set_ylabel(r'z (m)')
ax[0,0].legend()

ax[1,0].plot(rho_for1 - rho_analytical, z, label="dz = 500")
ax[1,0].plot(rho_for - rho_analytical, z, label="dz = 1000")
```

```
ax[1,0].set_title("Accuracy of forward ")
ax[1,0].set_ylabel("z")
ax[1,0].set_xlabel(r"$\rho_{scheme} - \rho_{analytical}$")
ax[1,0].set_xlim(-0.001,0)
ax[1,0].legend()


# Backward scheme
ax[0,1].plot(rho_back1, z, label="dz = 500")
ax[0,1].plot(rho_back, z, label="dz = 1000")
ax[0,1].plot(rho_analytical, z, color="red",linestyle="--",label='analytical')
ax[0,1].set_title("Backward Scheme")
ax[0,1].set_xlabel(r'$\rm \rho (g.cm^{-3})$')
ax[0,1].set_ylabel(r'z (m)')
ax[0,1].legend()

ax[1,1].plot(rho_back1 - rho_analytical, z, label="dz = 500")
ax[1,1].plot(rho_back - rho_analytical, z, label="dz = 1000")
ax[1,1].set_title("Accuracy of forward ")
ax[1,1].set_ylabel("z")
ax[1,1].set_xlabel(r"$\rho_{scheme} - \rho_{analytical}$")
ax[1,1].set_xlim(0,0.001)
ax[1,1].legend()

# Central scheme
ax[0,2].plot(rho_central1, z, label="dz = 500")
ax[0,2].plot(rho_central, z, label="dz = 1000")
ax[0,2].plot(rho_analytical, z, color="red",linestyle="--",label='analytical')
ax[0,2].set_title("Central Scheme")
ax[0,2].set_xlabel(r'$\rm \rho (g.cm^{-3})$')
ax[0,2].set_ylabel(r'z (m)')
ax[0,2].legend()

ax[1,2].plot(rho_central1 - rho_analytical, z, label="dz = 500")
ax[1,2].plot(rho_central - rho_analytical, z, label="dz = 1000")
ax[1,2].set_title("Accuracy of forward ")
ax[1,2].set_ylabel("z")
ax[1,2].set_xlabel(r"$\rho_{scheme} - \rho_{analytical}$")
ax[1,2].set_xlim(0,1e-5)
ax[1,2].legend()

plt.tight_layout()
plt.savefig("figures/exercise_figure/ex1_3_scheme_accur.png")
```

In this part, we try to compare the accuracy in case of dz = 500 and dz = 1000 using FDM.

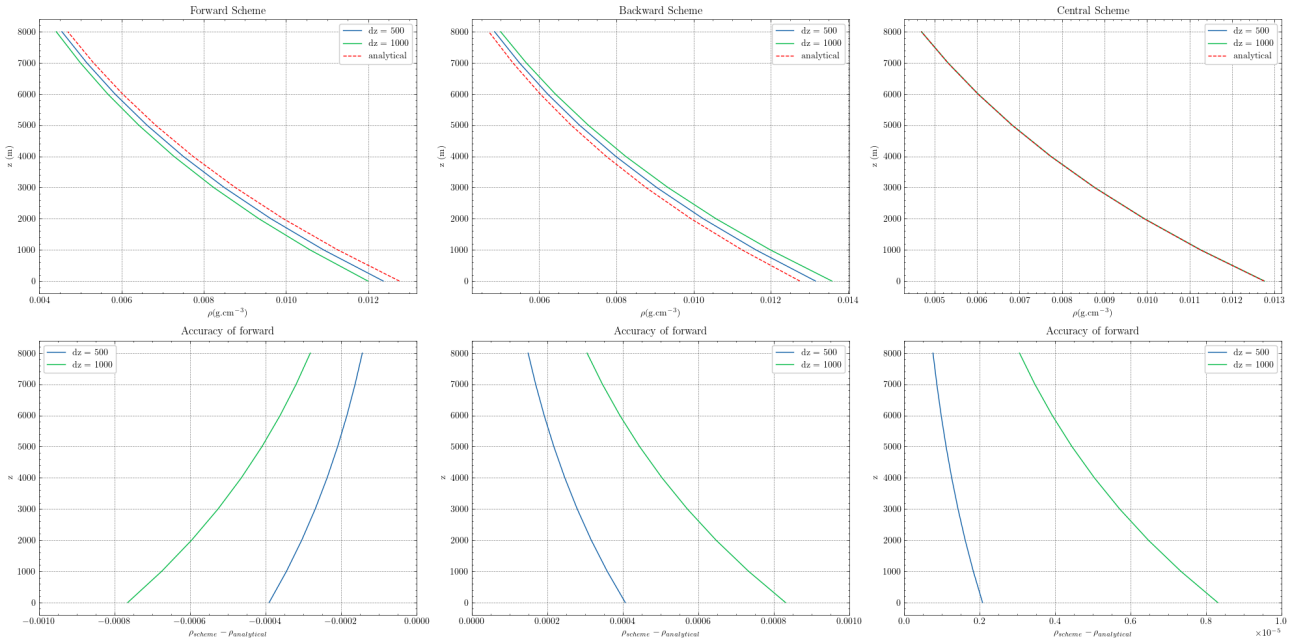


Figure 4: Comparing the accuracy of 3 schemes in dz = 500 and dz = 1000

From the figure 4, as the step get smaller, the value get closer to the analytical solution. And the accuracy also increase as the number of step becomes larger.

2.5 Atmospheric pressure

We have

$$\frac{dp}{dz} = -p_0/He^{-z/H} = -p(z)/H$$

So we can approximate the atmospheric pressure as follow:

$$p(z) = -H \times \frac{dp}{dz}$$

```
# Analytical
p_analytical = p(z)
# Forward
p_for = -H*(p(z+dz) - p(z))/dz
# Backward
p_back = -H*(p(z) - p(z-dz))/(dz)
# Center
p_central = -H*(p(z+dz/2) - p(z-dz/2))/(dz)
# 4-order
p_4order = H*(-2/3 * (p(z+dz) - p(z-dz))/(dz) + 1/12 * (p(z+2*dz) - p(z - 2*
    dz))/(dz))
```

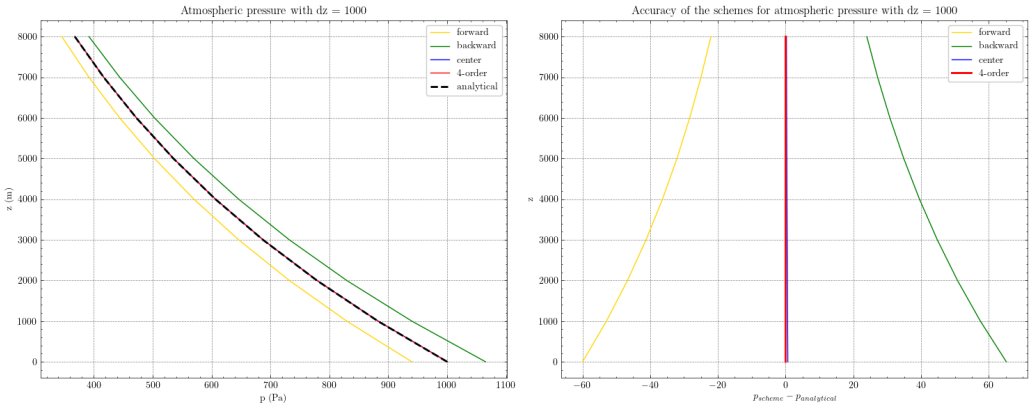


Figure 5: Atmospheric pressure

3 Taylor Expansion

Pratice#2.1:

1. Write a program to plot the analytical result of e^x for x=0, 0.1, ..., 2.
2. Write a program to calculate (approximately) the function ex for $0 \leq x < 2$ by using the Taylor's expansion
3. Compare with the analytical results (overlay on the plot)
4. Find out the truncation order for a good approximation (for $x < 1$; for $x < 2$ and so on)

Pratice#2.2:

1. Write a program to do a similar analysis for the approximation of sin(x)
2. Do a similar analysis for the approximation of 1/(1-x) for $x \neq 1$
3. Do a similar analysis for the approximation of $x^3 + x^2 + x + 1$

3.1 Taylor theorem

Taylor expansion is a method to approximate a function by an infinite sum of polynomial terms that are expressed in terms of the function's derivatives at a single point.

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + \frac{f^{(2)}(x)}{2!}(\Delta x)^2 + \frac{f^{(3)}(x)}{3!}(\Delta x)^3 + ... + \frac{f^{(n)}(x)}{n!}(\Delta x)^n + O(x^{n+1})$$

3.2 Exponential Function

```
x = np.arange(0,2,0.1)
f1 = np.exp(x)

def expfunc_taylor(n):
    f_tay = np.zeros_like(x)
    i = 0
    for i in range(n):
        f_tay = f_tay + (x)**i/m.factorial(i)
        i = i + 1
    return f_tay

plt.figure(figsize=(8,6))
plt.plot(x,f1, linestyle='--', linewidth=2,label='analytical', color="black"
    )
nlist = np.arange(2,6)
for i in nlist:
```

```
    plt.plot(x,expfunc_taylor(i), label=f'n = {i}')
plt.ylabel("x")
plt.xlabel(r"$e^{x}$")
plt.title(f"Exponential Function")
plt.legend()
plt.show()
```

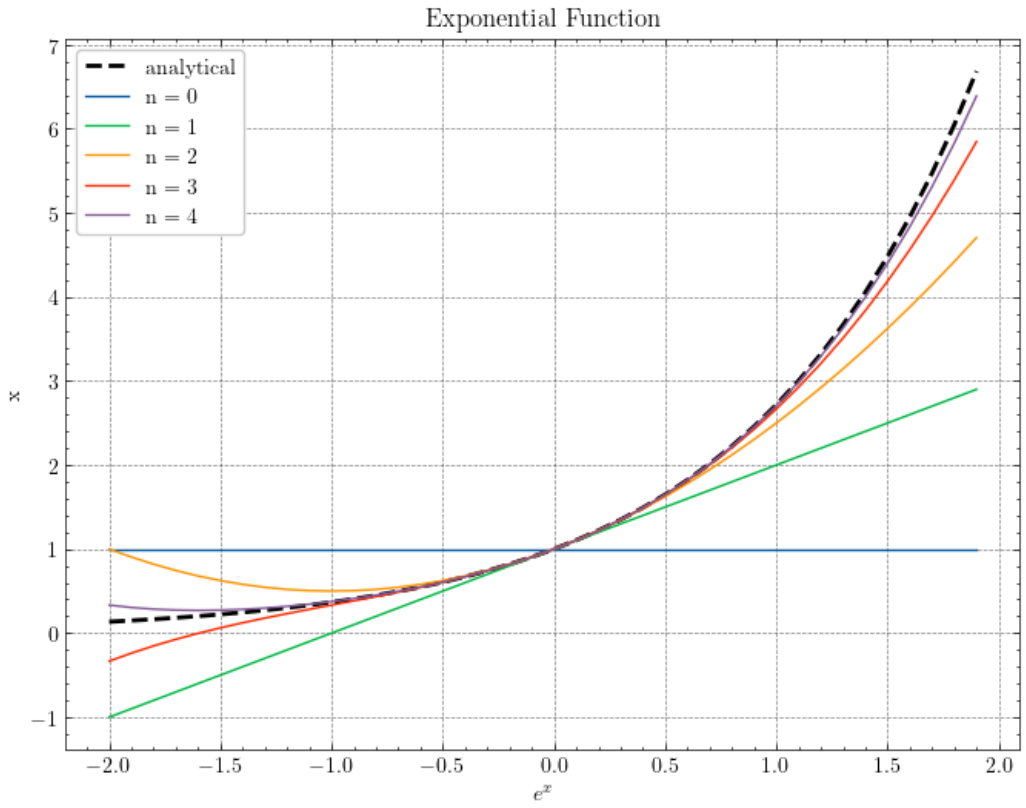


Figure 6: Exponential Function

Using the equation from section 3.1, we can approximate the exponential function e^x around x = 0 as:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

For the code, we just add the new term $x^n/n!$ at each order n to the taylor series as my function `expfunc_taylor`. From figure 6, we can see that the higher the order, the closer the value to the analytical solution.

3.3 Sinusoidal Function

```
x = np.arange(0,2,0.1)
f1 = np.sin(x)

def sin_taylor(n):
    f_tay = np.zeros_like(x)
    i = 1
    for i in range(n):
        f_tay = f_tay + (-1)**((i-1)/2) * (x)**i / m.factorial(i)
        i = i + 2
    return f_tay
plt.figure(figsize=(8,6))
plt.plot(x,f1,linestyle='--', linewidth=2, label='analytical', color="black"
    )
nlist = np.arange(2,10,2)
for i in nlist:
    plt.plot(x,sin_taylor(i), label=f'n = {i}')
plt.ylabel("x")
plt.xlabel(r"$sin(x)$")
plt.title(f"Sinusoidal Function")
plt.legend()
plt.show()
```

Similar to the exponential function, we can express the sinusoidal function around 0 as:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

For the code, we just add the term $(-1)^(\frac{n-1}{2})x^n/n!$ at each order n to the taylor series as my function `sin_taylor`. The sin function around 0 only contain odd-order terms, so we use the 1, 3, 5, 7 order.

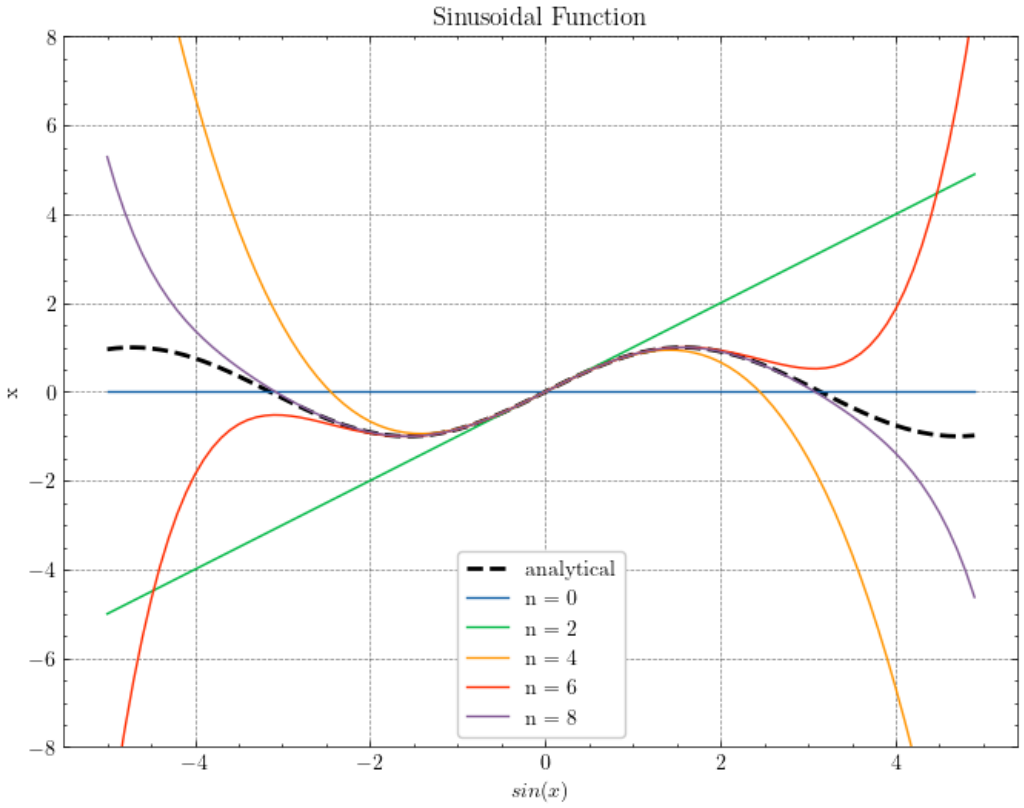


Figure 7: Sinusoidal Function

3.4 Geometric Function

```
x = np.arange(0,1,0.02)
f1 = 1/(1-x)

def inverse_tay(n):
    f_tay = np.zeros_like(x)
    i = 1
    for i in range(n):
        f_tay = f_tay + (x)**i
        i = i + 1
    return f_tay

plt.figure(figsize=(8,6))
plt.plot(x,f1,linestyle='--', linewidth=2, label='analytical', color="black"
    )
nlist = np.arange(2,10,2)
for i in nlist:
    plt.plot(x,inverse_tay(i), label=f'n = {i}')
plt.ylabel("x")
```

```
plt.xlabel(r"$\dfrac{1}{1-x}$")
# plt.xlim(0,1)
plt.title(f"1/(1-x) Function")
plt.legend()
plt.show()
```

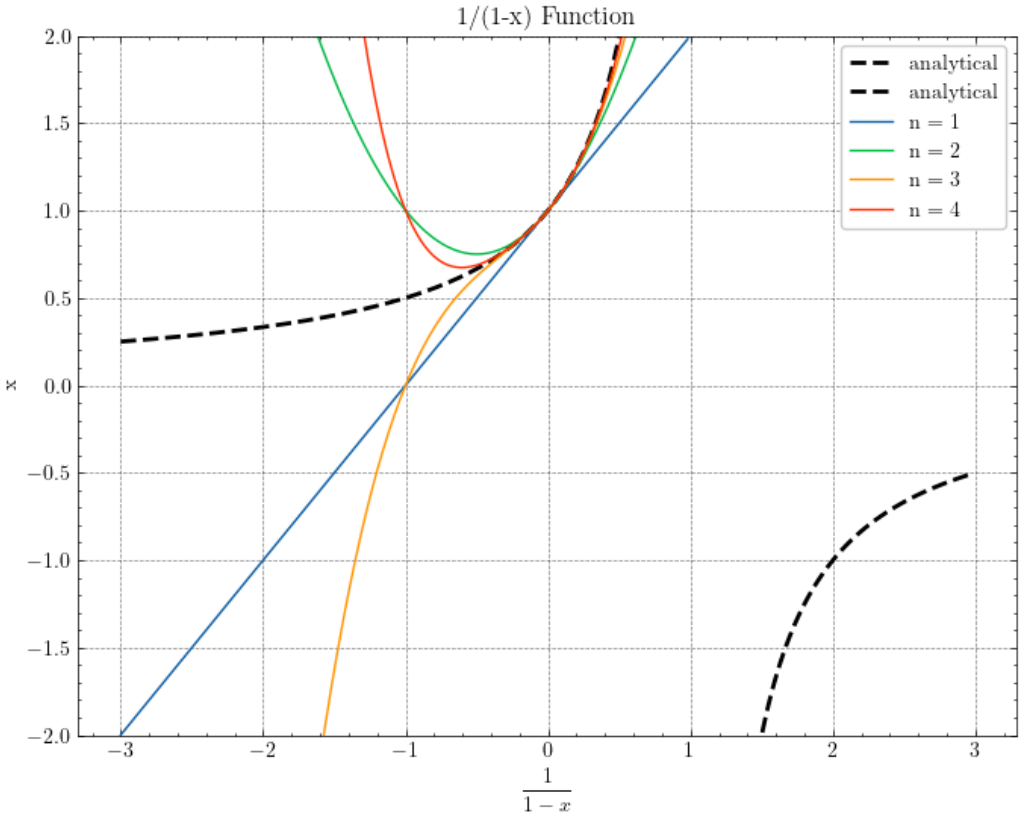


Figure 8: Geometric Function

The geometric function expands around x = 0:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$$

For the code, we just add the term x^n at each order n to the taylor series as my function `inverse_taylor`. Going higher in the order also lead to the higher accuraccy in approximation around x = 0 (except for the point in the domain x > 1). Because of the discontinuity at x = 1, we need to approximate the point in two seperate domain $(-\infty,1)$ and $(1,\infty)$.

3.5 Polynomial Function

```
x = np.arange(0,2,0.1)
f = lambda x: x**3 + x**2 + x + 1
f1 = lambda x: 3*x**2 + 2*x + 1
f2 = lambda x: 6*x + 2
f3 = 6

def poly_taylor(x, n ,x0 = 2):
    if n < 1:
        return 0
    elif n == 1:
        f_tay = f(x0) + f1(x0)*(x - x0)
    elif n == 2:
        f_tay = f(x0) + f1(x0)*(x - x0) + f2(x0)*(x-x0)**2/2
    else:
        f_tay = f(x0) + f1(x0)*(x - x0) + f2(x0)*(x-x0)**2/2 + f3*(x-x0)**3/
    m.factorial(3)
    return f_tay

plt.figure(figsize=(8,6))
plt.plot(x,f(x), linestyle='--', linewidth=2,label='analytical', color="
    black")
nlist = np.arange(1,4)
for n in nlist:
    plt.plot(x,poly_taylor(x, n), label=f'n = {n}')
plt.ylabel("x")
plt.xlabel(r"$x^3 + x^2 + x + 1$")
# plt.xlim(0,1)
plt.title(f"Polynomial Function")
plt.legend()
plt.show()
```

For this finite polynomial function, we can approximate the function around $x_0 = 2$
at 0-order:

$$f(x) = f(x_0) = 15$$

at 1-order:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) = -19 + 17x$$

at 2-order:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)(x - x_0^2}{2!} = 9 - 11x + 7x^2$$

at 3-order and higher order:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)(x - x_0^2}{2!} + \frac{f'''(x_0)(x - x_0)^3}{3!} = 1 + x + x^2 + x^3$$

At 3-order and higher order, the taylor series are exactly our original function.

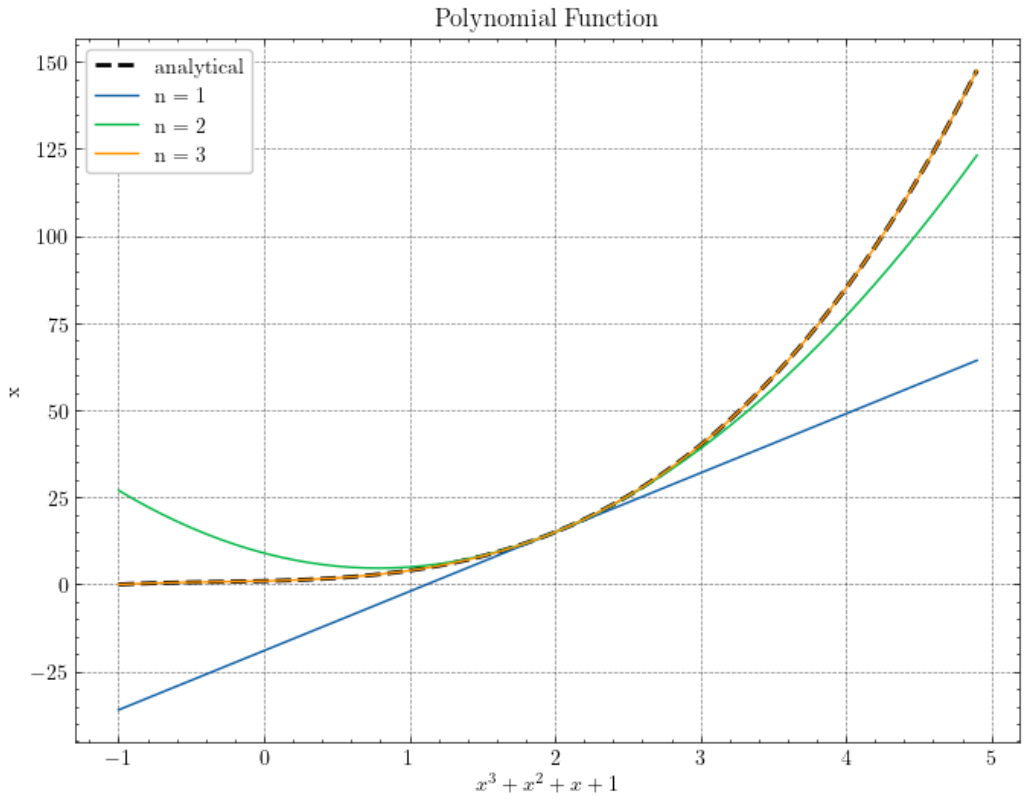


Figure 9: Polynomial Function

4 4-order accurate formulas

4.1 4-order accurate formulas

We approximate the first derivative using fourth-order accurate formulas:

$$f'(x) \approx \frac{2}{3}\frac{f(x+h) - f(x-h)}{h} - \frac{1}{12}\frac{f(x+2h) - f(x-2h)}{h}$$

Practice #3.1

$$f(x) = cos(x) + sin(2x) \quad -\pi \leq x \leq \pi$$

- compute the derivative f'(x) using:
- forward, backward, centered schemes
- 4th-order accurate formula
- Comparing the accuracy of the schemes

```
x = np.linspace(-np.pi, np.pi, 100)
delta_x = x[1] - x[0]

# function
f = lambda x: np.cos(x) + np.sin(2*x)

# derivative
f1 = lambda x: -np.sin(x) + 2*np.cos(2*x)

f_for = (f(x+delta_x) - f(x))/delta_x
f_back = (f(x) - f(x-delta_x))/delta_x
f_center = (f(x+delta_x) - f(x-delta_x))/(2*delta_x)
f_4th = 2/3 * (f(x+delta_x) - f(x-delta_x))/delta_x - 1/12 * (f(x+2*delta_x)
    - f(x-2*delta_x))/delta_x

# Plot
plt.figure(figsize=(12,8))
plt.plot(x, f1(x), linestyle='--', linewidth=2, label = "analytical", color=
    'black')
plt.plot(x, f_for, label = "forward")
plt.plot(x, f_back, label = 'backward')
plt.plot(x, f_center, label = 'center')
plt.plot(x, f_4th, label = '4-order')

plt.xlabel("x")
plt.ylabel(r"$\partial f / \partial x$")
plt.title(r"$f(x) = cos(x) + sin(2x)$")
plt.legend()
```

```
 # location for the zoomed portion
sub_axes = plt.axes([.6, .6, .25, .25])

# # plot the zoomed portion
x_detail = x[50:60]
sub_axes.plot(x_detail, f1(x_detail), linestyle='--', linewidth=2, label = "
    analytical", color='black')
sub_axes.plot(x_detail, f_for[50:60])
sub_axes.plot(x_detail, f_back[50:60])
sub_axes.plot(x_detail, f_4th[50:60])

plt.show()
```

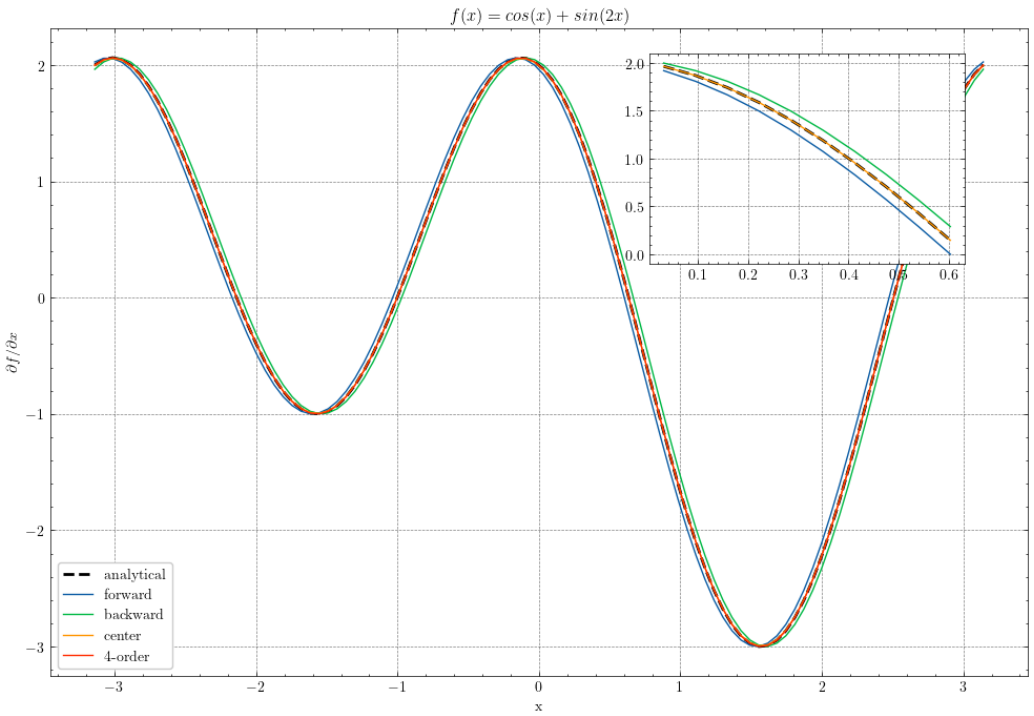


Figure 10: Derivative of f(x0 = cos(x) + sin(2x) with analytical, forward, backward, center and 4-order schemes

We can see that every scheme are pretty good of approximating the function, despite some offsets in the forward and backward schemes (because they are first order error). Center and fourth order scheme are not distinguishable from analytical curve in this plot.

```
plt.figure(figsize=(12,8))
plt.plot(x, f_for - f1(x), label = 'Forward')
plt.plot(x, f_back - f1(x), label = "Backward")
plt.plot(x, f_center - f1(x), label = "Center")
plt.plot(x, f_4th - f1(x), label = "4-order")

plt.xlabel("x")
plt.ylabel(r"$f_x^{scheme} - f_x^{analytical}$")
plt.title("Accuracy of the schemes")
plt.legend()
plt.show()
```

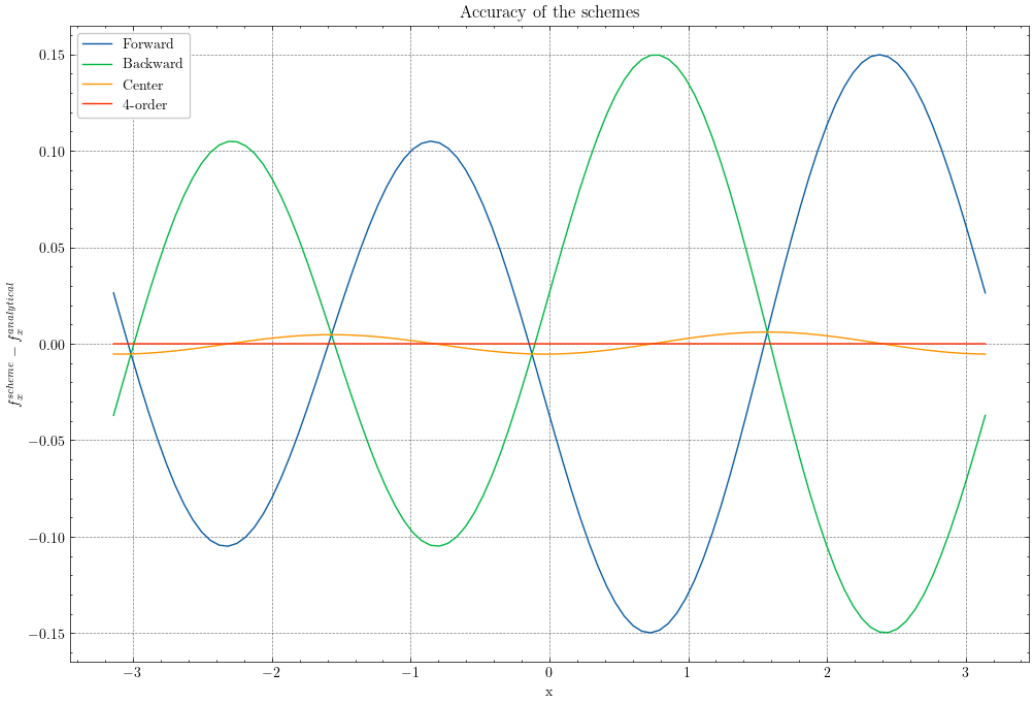


Figure 11: Accuracy of 4 schemes compared to analytical solution. The closer the value to 0, the more accurate the scheme

However, from the accuracy plot, the fourth order error scheme has a higher accuracy than center scheme as the value is nearly zero.

Practice #3.2

Revisit Practice#1: Vertical structure of the atmosphere

- Write a program to calculate (approximately) the air density at different atmospheric levels by using forward, backward, centered difference methods, etc., given the pressure:

$$p(z) = p_0 e^{-z/H}$$

  where H=8000(m); g=9.8 m/s2; p0=1000.; Z=0., 1000., ...9000.; dz=1000.

- Compare the obtained simulation results with the analytical results.
- Use a 4-order difference approximation to solve the problem

```
# Initial conditions
H = 8000
p0 = 1000
dz = 1000
z = np.arange(0,9000,dz)
g = 9.8

# Atmospheric Pressure
p = lambda z: p0*np.exp(-z/H)
dp = lambda z: -p0/H * np.exp(-z/H)

# 4-order scheme
dp_ana = dp(z)
dp_4th = 2/3 * (p(z+dz) - p(z-dz))/dz - 1/12 * (p(z+2*dz) - p(z-2*dz))/dz

#Plot
fig, (ax1, ax2) = plt.subplots(1,2,figsize=(15,6))
ax1.plot(dp_4th,z, color='black',linewidth=2,label="4-order")
ax1.plot(dp_ana,z,"r--",linewidth=2,label='analytical')

ax1.set_xlabel("dp/dz")
ax1.set_ylabel("z")
ax1.legend()

ax2.plot(dp_4th - dp_ana, z,label = 'Accuracy of 4-order scheme')
ax2.set_xlim(-1e-5,1e-5)

ax2.set_xlabel("$dp_{4-th}/dz - dp_{ana}/dz$")
ax2.set_ylabel("z")
ax2.legend()

plt.tight_layout()
plt.show()
```

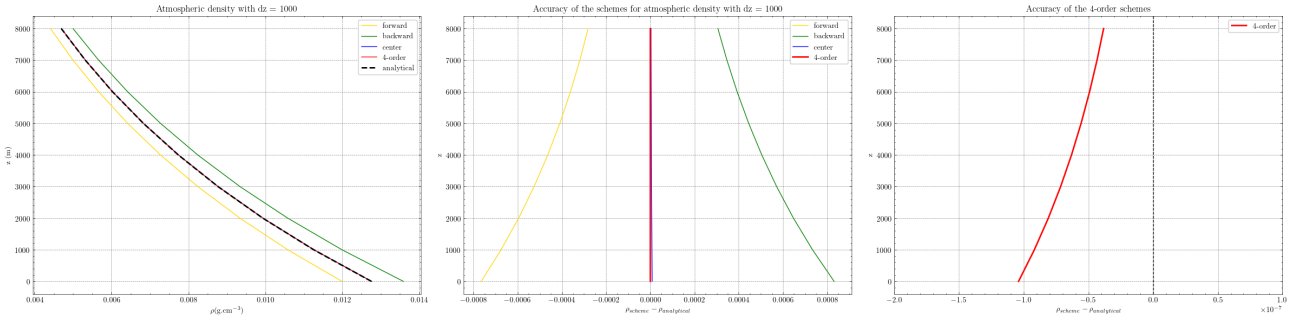
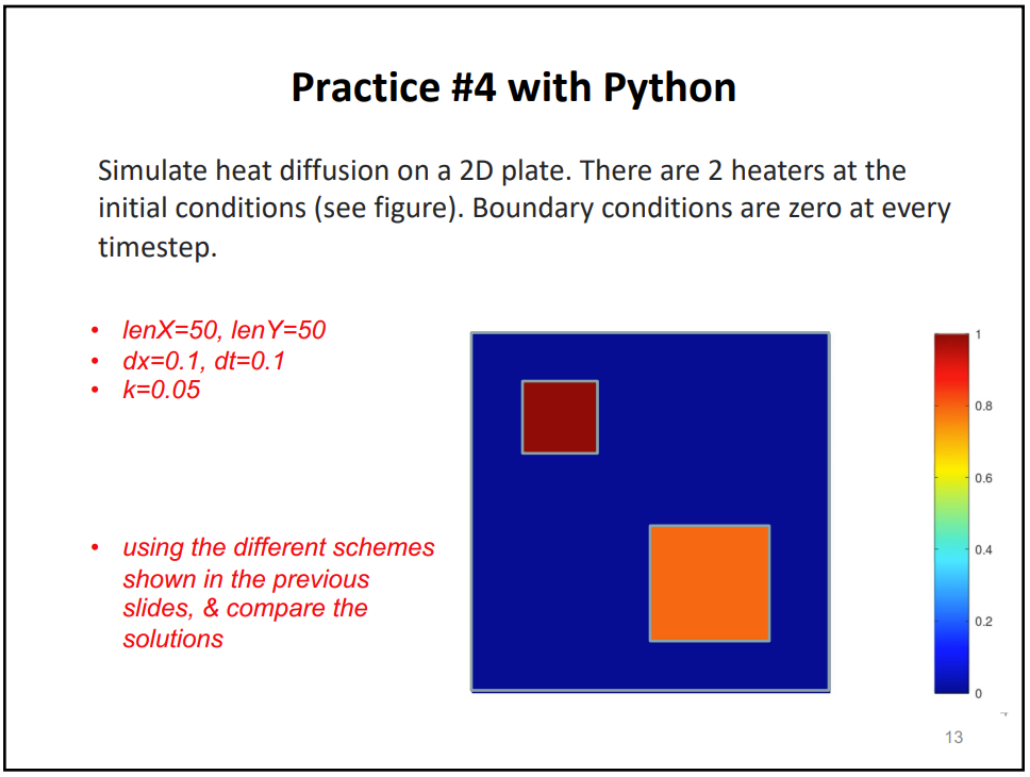


Figure 12: Derivative of atmospheric pressure by 4-order scheme

From figure 12, we can see that the 4-order accurate scheme approximate the value with the error at around 10^{-7}, this is a very high precision compared to other schemes like forward, backward and centered.

5 Heat Transfer



5.1 Second-order accurate Laplacian

5-point diamond stencil

$$\nabla^2\psi(i,j) \cong \frac{\psi(i+1,j)+\psi(i-1,j)+\psi(i,j+1)+\psi(i,j-1)-4\psi(i,j)}{\Delta^2} + O(\nabla)^2$$

5-point square stencil

$$\nabla^2\psi(i,j) \cong \frac{\psi(i+1,j+1)+\psi(i-1,j-1)+\psi(i-1,j+1)+\psi(i+1,j-1)-4\psi(i,j)}{2\Delta^2} + O(\nabla)^2$$

9-point stencil

$$\begin{aligned}\nabla^2\psi(i,j) \cong & \frac{1}{6\Delta^2}\{\psi(i+1,j+1)+\psi(i-1,j-1)+\psi(i-1,j+1)+\psi(i+1,j-1) \\ & + 4[\psi(i+1,j)+\psi(i-1,j)+\psi(i,j+1)+\psi(i,j-1)] - 20\psi(i,j)\} + O(\nabla)^2\end{aligned}$$

5.2 Fourth-order accurate Laplacian

9-point square

$$\nabla^2\psi(i,j) \cong \frac{1}{2\Delta^2}\{-[\psi(i+1,j+1)+\psi(i-1,j-1)+\psi(i-1,j+1)+\psi(i+1,j-1)] + 4[\psi(i+1,j)+\psi(i-1,j)+\psi(i,j+1)+\psi(i,j-1)] - 12\psi(i,j)\} + O(\nabla)^4$$

9-point pinwheel

$$\nabla^2\psi(i,j) \cong \frac{1}{12\Delta^2}\{16[\psi(i+1,j+1)+\psi(i-1,j-1)+\psi(i-1,j+1)+\psi(i+1,j-1)] - [\psi(i,j+2)+\psi(i-2,j)+\psi(i,j-2)+\psi(i+2,j)] - 60\psi(i,j)\} + O(\nabla)^4$$

5.3 Code

```
import numpy as np

dt = 0.1
dx = 0.1
k = 0.05

def square_5point(ini_heat):
    heat = np.zeros((500,500))
    for j in range(1, 499):
        for i in range(1, 499):
            heat[i,j] = ini_heat[i,j] + dt*k*(ini_heat[i+1,j+1] + ini_heat[i
   -1,j-1] + ini_heat[i-1,j+1] + ini_heat[i+1,j-1] - 4*ini_heat[i,j])/(2*dx
   **2)
    return heat

def diamond_5point(ini_heat):
    heat = np.zeros((500,500))
    for j in range(1, 499):
        for i in range(1, 499):
            heat[i,j] = ini_heat[i,j] + dt*k*(ini_heat[i+1,j] + ini_heat[i
   -1,j] + ini_heat[i,j+1] + ini_heat[i,j-1] - 4*ini_heat[i,j])/(dx**2)
    return heat

def square_9point_2order(ini_heat):
    heat = np.zeros((500,500))
    for j in range(1, 499):
        for i in range(1, 499):
            heat[i,j] = ini_heat[i,j] + dt*k*(
                ini_heat[i+1,j+1] + ini_heat[i-1,j-1] + ini_heat[i-1,j+1] +
   ini_heat[i+1,j-1]
```

```
                + 4*(ini_heat[i+1,j] + ini_heat[i-1,j] + ini_heat[i,j+1] + ini_heat[i,j-1])
                - 20*ini_heat[i,j]
                                                )/(6*dx**2)
    return heat

def square_9point_4order(ini_heat):
    heat = np.zeros((500,500))
    for j in range(1, 499):
        for i in range(1, 499):
            heat[i,j] = ini_heat[i,j] + dt*k*(
                - (ini_heat[i+1,j+1] + ini_heat[i-1,j-1] + ini_heat[i-1,j+1] + ini_heat[i+1,j-1])
                + 4*(ini_heat[i+1,j] + ini_heat[i-1,j] + ini_heat[i,j+1] + ini_heat[i,j-1])
                - 12*ini_heat[i,j]
            )/(2*dx**2)
    return heat

def pinwheel_9point_4order(ini_heat):
    heat = np.zeros((500,500))
    for j in range(2, 498):
        for i in range(2, 498):
            heat[i,j] = ini_heat[i,j] + dt*k*(
                - (ini_heat[i+1,j+1] + ini_heat[i-1,j-1] + ini_heat[i-1,j+1] + ini_heat[i+1,j-1])
                + 4*(ini_heat[i+1,j] + ini_heat[i-1,j] + ini_heat[i,j+1] + ini_heat[i,j-1])
                - 20*ini_heat[i,j]
            )/(2*dx**2)
    return heat
```

Listing 1: Python file for defining schemes, spatial domain and time interval

```
import os
import numpy as np
import matplotlib.pyplot as plt
import scienceplots
from tqdm import tqdm
from schemes_ import *

plt.style.use(['science'])

# File path
data_path = 'data/diamond-5point'
figure_path = 'figures/diamond-5point'

# Time
t_max = 500
```

```
steps = int(t_max / dt)

# Square initialization
ini_heat = np.zeros((500, 500))
ini_heat[100:180, 300:380] = 1
ini_heat[250:400, 100:250] = 0.8

# Initial file
heat = np.copy(ini_heat)
np.save(os.path.join(data_path,'heat_diffuse_0000'), ini_heat)

# Create a progress bar
progress_bar = tqdm(total=steps, unit=' Steps', ncols=100, ascii=True)

for t in range(steps):
    ini_heat = np.copy(heat)
    # Change the schemes
    heat = diamond_5point(ini_heat)
    if t % 10 == 0:
        np.save(os.path.join(data_path,f'heat_diffuse_{t:05d}'), heat)
        plt.figure(figsize=(8, 7))
        plt.imshow(ini_heat, origin='lower', cmap='jet', vmin=0, vmax=1)

        # Colorbar with label
        cbar = plt.colorbar(fraction=0.046, pad=0.04)
        cbar.set_label('Temperature')

        plt.minorticks_on()
        plt.tick_params(axis='both', which='both', direction='out', top=
   False, right=False)
        # Labels for x and y axes
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title(f"Time = {np.round(t*dt,2)}")

        plt.savefig(os.path.join(figure_path,f'heat_diffuse_{t:05d}'))
        plt.close()
    else:
        pass
    # Update the progress bar
    progress_bar.update(1)
```

Listing 2: Code for running the heat diffusion and save the result for each 10 step

5.4 Result

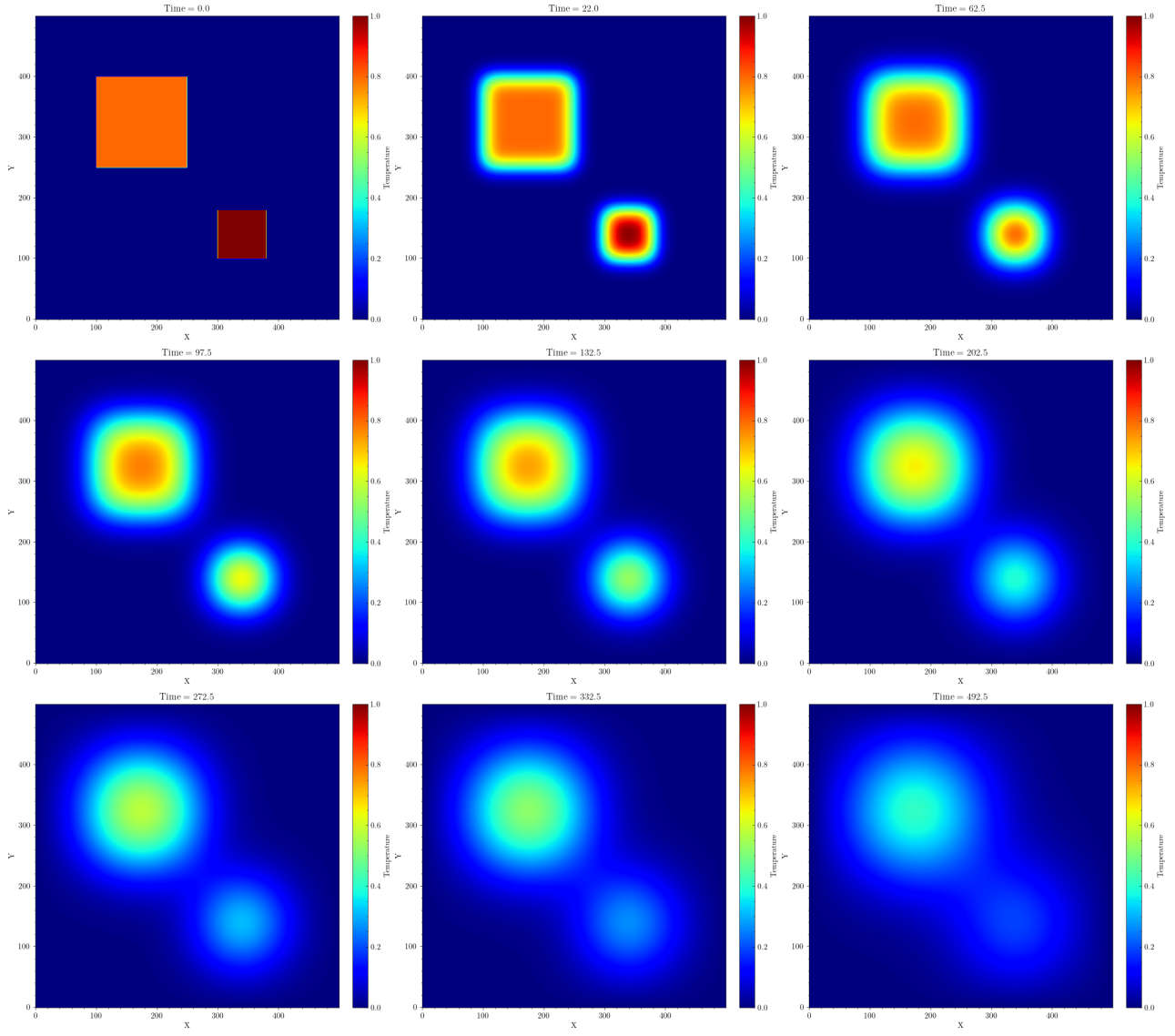


Figure 13: Heat Diffusion

Dicussion:

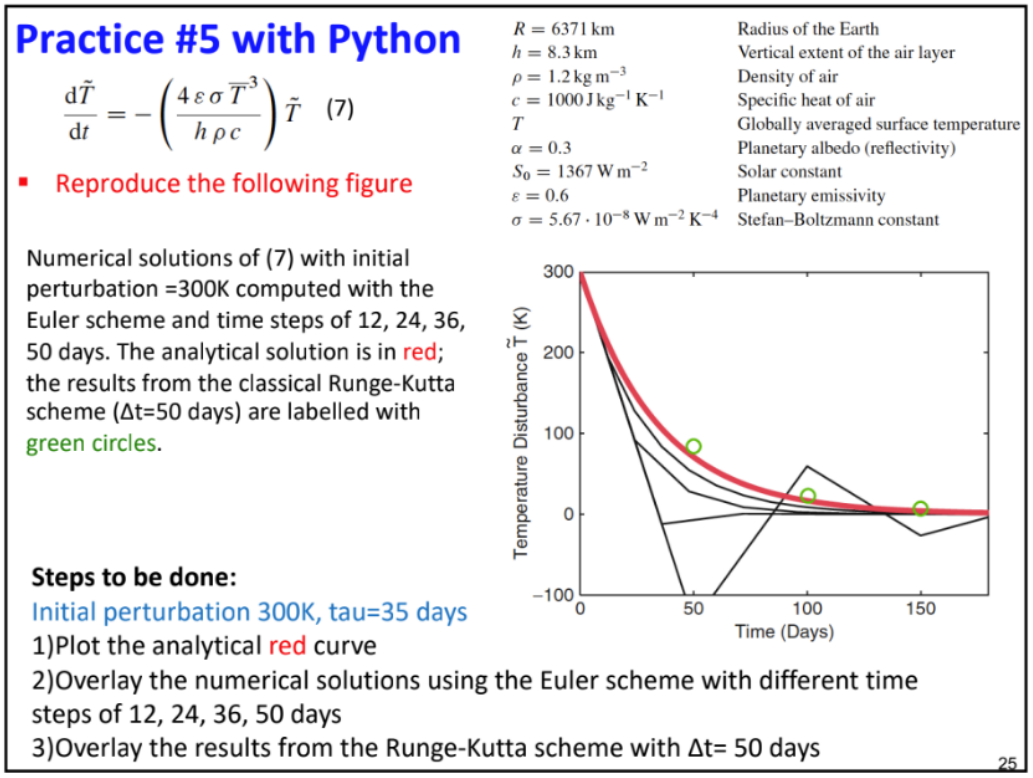
In the code section, there are two python files that should be saved in the same directory. The first python file named `scheme_.py`, that will be called in, at the beginning of the second file. The `scheme.py` file saved all the function for defining square 5-point, diamond 5-point, square 9-point, square 9-point 4-order and pinwheel 9-point 4-order, also defining spatial and temporal interval for numerical implementation in second file. The second file defines the initial condition

and contains for loop and progression bar for running and monitor the run. I also save the file in `.npy` format and plot it on each run and save it to corresponding figure directory.

For choosing the time interval, all the schemes can give the above result, but only the 5-point diamond scheme ran with dt = 0.1, the others cracked. Decreasing the time interval dt = 0.05 or 0.01 makes the others schemes stable.

At the beginning, the heat diffuses very fast because of the shape difference between places. As time goes on, the heat becomes more homogenous, so the diffusion becomes slower.

6 Numerical solution of an ordinary differential equation of 1st order



6.1 Analytical and numerical methods

A linear, homogenous differential equation of 1st order for the temperature perturbation:

$$\frac{d\widetilde{T}}{dt} = -\frac{4\epsilon\sigma\overline{T}^3}{h\rho c}\widetilde{T}$$

The analytical solution can be written as:

$$\widetilde{T} = \widetilde{T}(0)e^{-t/\tau} \tag{2}$$

where $\widetilde{T}(0)$ is the initial temperature pertubation, $\tau = \dfrac{h\rho c}{4\epsilon\sigma\overline{T}^3}$ is the time constant, indicate how rapidly an exponential function decays.

For the numerical method, we will use the forward scheme with the time interval $\Delta t = 12$, 24,

36, 50:

$$\frac{\widetilde{T}(i)-\widetilde{T}(i-1)}{\Delta t}=-\frac{1}{\tau}\widetilde{T}$$
$$\implies \quad \widetilde{T}(i)=(1-\frac{\Delta t}{\tau})\widetilde{T}(i-1)$$

Fourth-order Runge Kutta

The numerical solution using the fourth-order Runge-Kutta method can be expressed iteratively as follows:

$$\begin{aligned}
k_1 &= -\frac{1}{\tau}\widetilde{T}(i-1)\\
k_2 &= -\frac{1}{\tau}\left(\widetilde{T}(i-1)+\frac{\Delta t}{2}k_1\right)\\
k_3 &= -\frac{1}{\tau}\left(\widetilde{T}(i-1)+\frac{\Delta t}{2}k_2\right)\\
k_4 &= -\frac{1}{\tau}\left(\widetilde{T}(i-1)+\Delta t k_3\right)\\
\implies \quad \widetilde{T}(i) &= \widetilde{T}(i-1)+\frac{\Delta t}{6}(k_1+2k_2+2k_3+k_4)
\end{aligned}$$

Here, $\widetilde{T}(i-1)$ is the temperature perturbation at the previous time step, Δt is the time interval, and τ is the time constant.

It's worth noting that the fourth-order Runge-Kutta method provides a more accurate approximation compared to simpler numerical schemes. In our problem we will use the Runge Kutta with $\Delta x = 12$

6.2 Code

```
import numpy as np
import scienceplots
import matplotlib.pyplot as plt
import astropy.constants as c
import astropy.units as u

plt.style.use(['science','grid'])

# Constants
sigma_sb = c.sigma_sb
```

```
epsilon = 0.6
h = 8.3 * u.km
rho = 1.2 * u.kg / u.m**3
c0 = 1000 * u.J / (u.kg*u.K)
T_avg = 300 * u.K
tau = (h * rho * c0) / (4 * epsilon * sigma_sb * T_avg**3)
tau = tau.to(u.day).value

# Analytical
T_per = lambda t : T_avg*np.exp(-t/tau)

# Numerical
def T_numerical(delta_t):
    t_max = 200
    t_step = int(t_max/delta_t)
    T_num = np.empty(t_step + 1)
    time = np.empty(t_step + 1)
    T_num[0] = T_avg.value
    time[0] = 0
    for i in range(1, t_step + 1):
        T_num[i] = (1 - delta_t/tau)*T_num[i - 1]
        time[i] = time[i-1] + delta_t
    return time, T_num

time12, T_num12 = T_numerical(12)
time24, T_num24 = T_numerical(24)
time36, T_num36 = T_numerical(36)
time50, T_num50 = T_numerical(50)

def runge_kutta(delta_t):
    fy = lambda y: -1/tau*y
    t_max = 200
    t_step = int(t_max/delta_t)
    T_num = np.empty(t_step + 1)
    time = np.empty(t_step + 1)
    T_num[0] = T_avg.value
    time[0] = 0
    for i in range(1, t_step + 1):
        k1 = delta_t * fy(T_num[i - 1])
        k2 = delta_t * fy(T_num[i - 1] + k1/2)
        k3 = delta_t * fy(T_num[i - 1] + k2/2)
        k4 = delta_t * fy(T_num[i - 1] + k3)
        T_num[i] = T_num[i-1] + (k1 + 2*k2 + 2*k3 + k4)/6

        time[i] = time[i-1] + delta_t
    return time, T_num

# Plot
fig, (ax1, ax2) = plt.subplots(1,2,figsize = (15,6))
```

```
ax1.plot(time12, T_per(time12), label="Analytical", color = 'black',
    linewidth=2)

ax1.plot(time50, T_num50, label=r"$\Delta_t = $50 days")
ax1.plot(time36, T_num36, label=r"$\Delta_t = $36 days")
ax1.plot(time24, T_num24, label=r"$\Delta_t = $24 days")
ax1.plot(time12, T_num12, label=r"$\Delta_t = $12 days")

delta_t = 12
time_runge, T_runge = runge_kutta(delta_t)
ax1.scatter(time_runge, T_runge, label=f"Runge-Kutta: $\Delta_t = $ {delta_t
    } days", marker='o', c = 'purple')

ax1.set_xlabel("Time (days)")
ax1.set_ylabel("Temperature (K)")
ax1.set_xlim(0, 200)

ax1.set_title("1D Temperature Model with Clouds")
ax1.legend()

ax2.plot(time50, T_num50 - T_per(time50).value, label=r"$\Delta_t = $50 days
    ")
ax2.plot(time36, T_num36 - T_per(time36).value, label=r"$\Delta_t = $36 days
    ")
ax2.plot(time24, T_num24 - T_per(time24).value, label=r"$\Delta_t = $24 days
    ")
ax2.plot(time12, T_num12 - T_per(time12).value, label=r"$\Delta_t = $12 days
    ")
ax2.plot(time_runge, T_runge - T_per(time_runge).value, label=f"Runge-Kutta:
    $\Delta_t = $ {delta_t} days",c='purple')
ax2.set_title("Accuracy")
ax2.set_xlabel("Time (days)")
ax2.set_ylabel("Temperature (K)")
ax2.legend()
plt.tight_layout()

plt.savefig("figures/exercise_figure/ex5")
```

6.3 Result

From the figure 14, we can see that with the forward scheme, the smaller the time interval, the more accurate the solution. At big time step like $\Delta t = 36$ or $\Delta t = 50$, the forward scheme even give the negative value of temperature in Kelvin which do not have any physical meanings. At the same $\Delta t = 12$, Runge Kutta scheme gave a better approximation than forward scheme as expected when we compare between first-order accuracy and fourth-order accuracy. The Runge

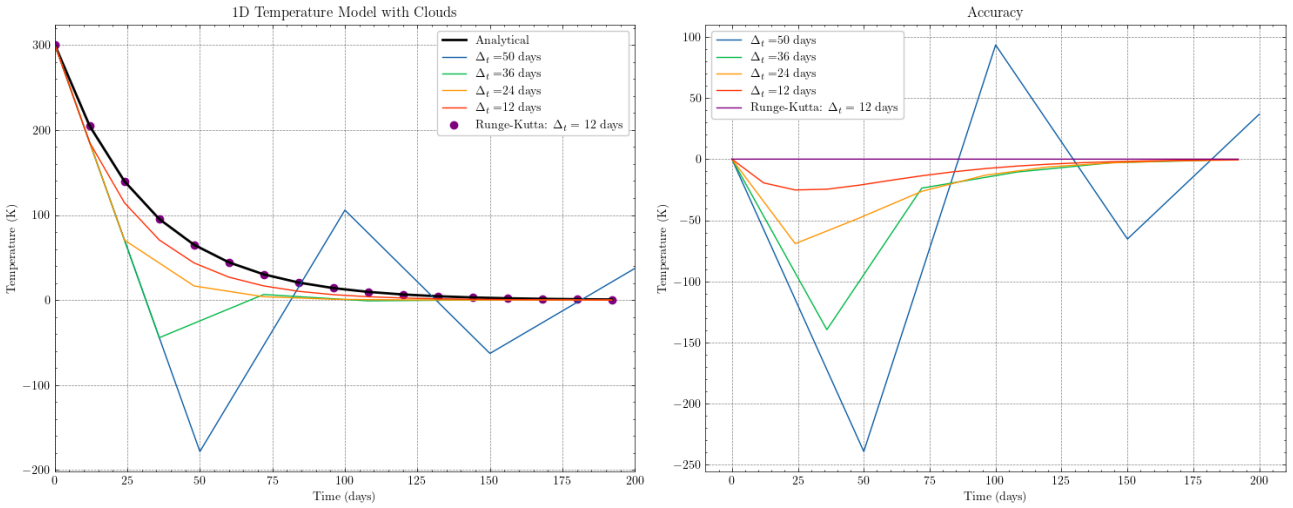


Figure 14: 1D Temperature Model with Clouds

Kutta fits the analytical curve very good, the error is nearly indistinguishable as we can observe on the right figure.

7 1D Diffusion Equation

- Solve the 1D heat equation with the initial and boundary condition:

$$u(x,0) = f(x) \quad u(0,t) = 0 \quad u(L,t) = 0$$

- with the initial heat distribution $f(x) = 6sin(\frac{\pi x}{L})$
- k=1
- L=1, T=0.1
- ns=11 is the amount of space points
- nt=31 is the amount of time steps

1. Plot figures for the FTCS scheme
2. Compare with the analytical solution?
3. Plot similar figures for ns=21 à explain why?

7.1 Analytical and Numerical Methods for Diffusion equation

7.1.1 Diffusion

General equation for Diffusion in 1D:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$$

Consider a rod with the length L with sinusoidal intial temperature u(x,0):

$$u(x,0) = f(x) = 6sin(\frac{\pi x}{L})$$

and Dirichlet boundary conditions, meaning that the values at two ends do not change and always equal to 0

$$u(0,t) = 0$$
$$u(L,t) = 0$$

The analytical seperable solution for this diffusion problem is expressed as:

$$u(x,t) = 6sin(\frac{\pi x}{L})e^{-k(\frac{\pi x}{L})^2 t}$$

7.1.2 Forward in time, centered in space (FTCS)

We discretized the spatial domain x from (0, L) into m interval denoted as x_i where i $\in$ 0, 1, 2, ..., m. We discretized the time domain x from (0, t_{max}) into n interval denoted as x_i where j $\in$ 0, 1, 2, ..., n. The FTCS scheme can be expressed as:

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = k\frac{u_{i+1}^j u_{i-1}^j - 2u_i^j}{\Delta x^2}$$
$$\implies \quad u_i^{j+1} = (1 - 2\alpha)u_i^j + \alpha(u_{i+1}^j + u_{i-1}^j)$$

where $\alpha = k\frac{\Delta t}{\Delta x^2}$.

The stability condition or CFL condition can be expressed as

$$\alpha \leq \frac{1}{2}$$
$$\implies \quad \Delta t \leq \frac{1}{2}\frac{\Delta x^2}{k}$$

NS = 11

```
L = 1; k = 1; T = 0.1; ns = 11; nt = 31

f = lambda x: 6*np.sin(np.pi*x/L)
x = np.linspace(0,L,ns)
t = np.linspace(0,T,nt)

# Analytical solution
u = lambda x,t: 6*np.sin(np.pi*x/L)*np.exp(-k*(np.pi/L)**2*t)
xv, tv = np.meshgrid(x,t)
heat_ana = u(xv,tv)

# FTCS scheme
ini_heat = f(x)

delta_x = x[1] - x[0]
delta_t = t[1] - t[0]
alpha = k*delta_t/delta_x**2

heat = np.zeros((len(t),(len(x))))
heat[0,:] = ini_heat

for j in range(1,len(t)):
    ini_heat = heat[j-1,:]
    for i in range(1, len(x)-1):
        heat[j,i] = (1 - 2*alpha) * ini_heat[i] + alpha*(ini_heat[i+1] +
   ini_heat[i-1])
```

```
26
27 # Plot
28 plt.figure(figsize=(12,10))
29
30 cmap = plt.get_cmap('jet')
31
32 for i,time in enumerate(t):
33     color = cmap(i / len(t))
34     plt.plot(x, heat[i,:], label = f"t = {np.round(time,3)}", color = color)
35
36 plt.xlabel("x")
37 plt.ylabel("u(x,t)")
38 plt.title("FTCS Scheme")
39 plt.legend()
40 plt.savefig("figures/exercise_figure/ex7_curve_ns11")
```

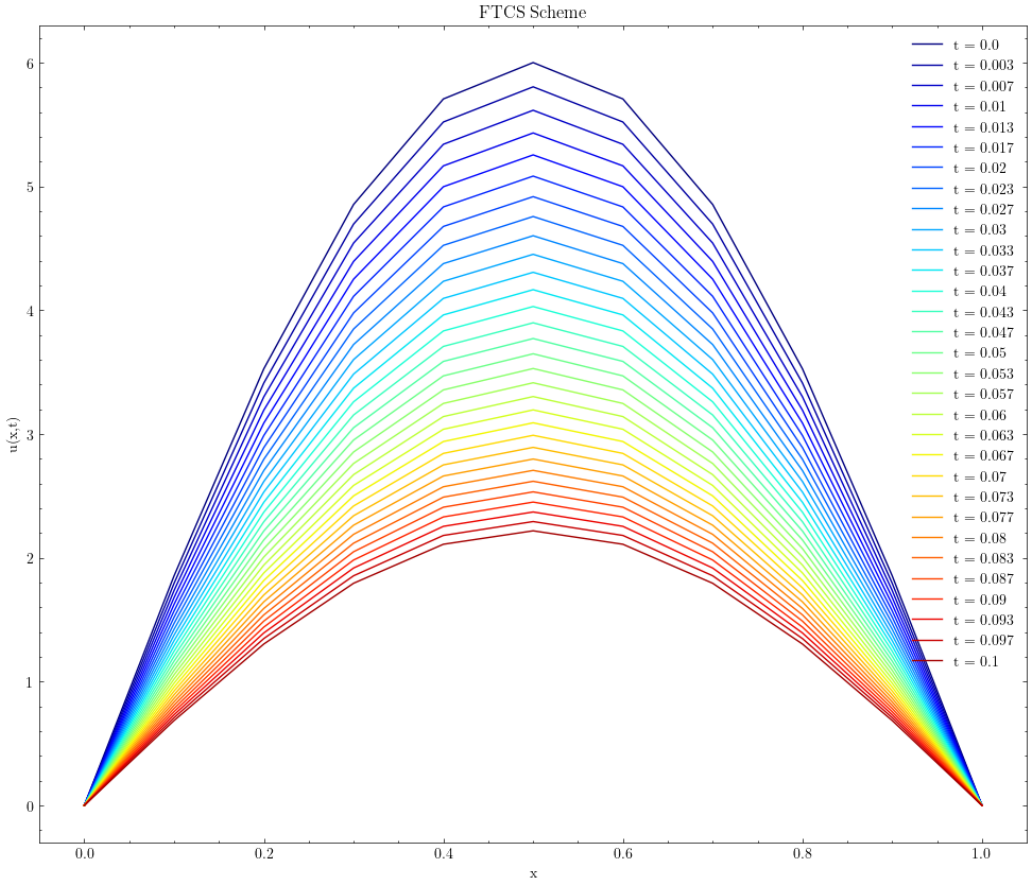


Figure 15: FTCS Scheme - Curve

```
plt.figure(figsize=(12,10))
plt.imshow(heat.T, origin='lower',cmap='jet')

plt.xlabel("t")
plt.ylabel("x")

plt.title("FTCS Scheme - ns = 11")
cbar = plt.colorbar(fraction=0.017, pad=0.03)
cbar.set_label('u(x,t)')
plt.tick_params(axis='both', which='both', direction='out', top=False, right
    =False)
plt.savefig("figures/exercise_figure/ex7_heatmap_ns31")
```

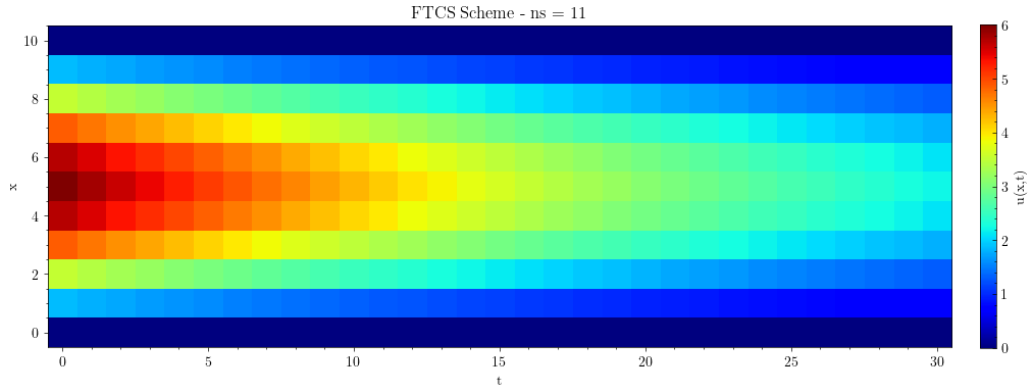


Figure 16: FTCS Scheme - Heatmap

```
plt.figure(figsize=(12,10))
plt.imshow(heat.T - heat_ana.T, origin='lower',cmap='jet')

plt.xlabel("t")
plt.ylabel("x")

cbar = plt.colorbar(fraction=0.017, pad=0.03)
cbar.set_label('$u_{FTCS} - u_{ana}$')
plt.savefig("figures/exercise_figure/ex7_compare_FTCS_analytical")
```

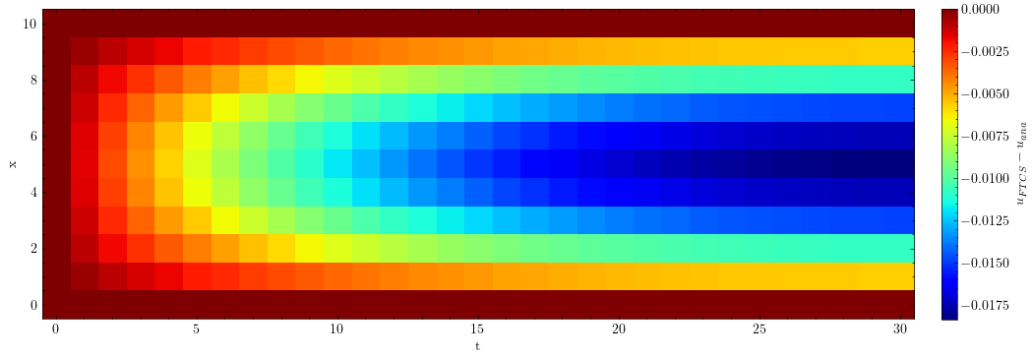


Figure 17: FTCS Scheme compares with analytical solution

From the figure 15 and 16, we can see that as the time goes on the heat have the tendency is to decrease to 0 as the rod is cooling down to the temperature at the boundary condition. The rate of temperature change depends on the temperature difference, so the rate of change is slower as the time goes on and the center of the rod cools down the fastest.

This numerical solution is stable because with ns $= 11$ and nt $= 31$, we have $\Delta x = 0.1$ and $\Delta t = 0.00333$, so stable condition $\alpha = 0.333 \leq 0.5$.

Comparing the heatmap of FTCS scheme for diffusion equation with analytical solution, from the figure 17, we notice that, overall the soluion of FTCS scheme is lower than analytical solution. Also, as time goes on, the numerical solution deviate more and more from the analytical solution. The center deviates the fastest, the trend of deviation decrease towards two ends.

NS = 21

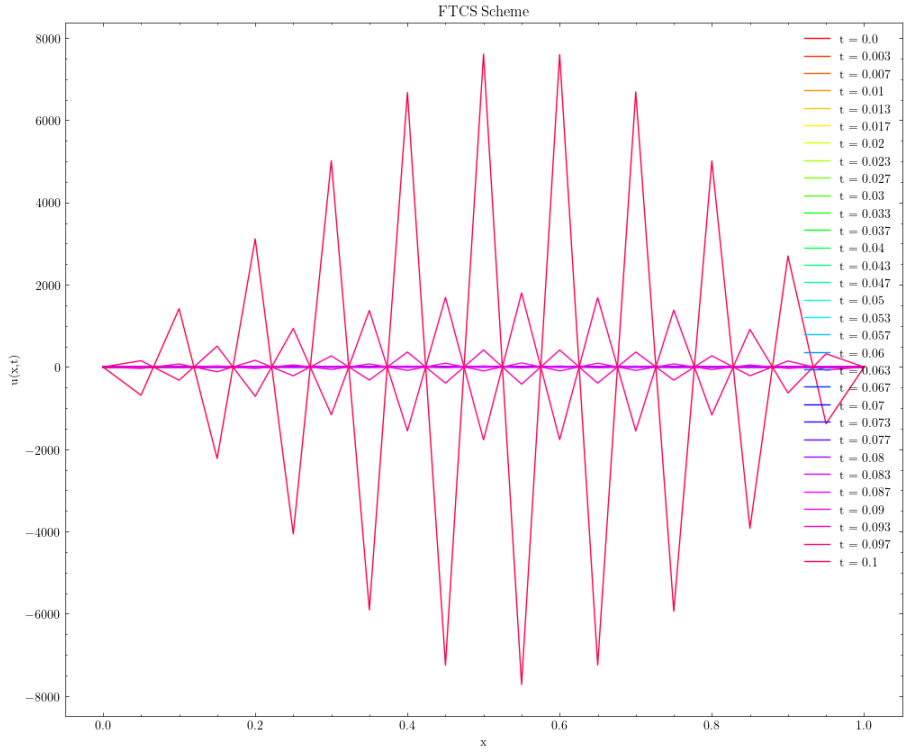


Figure 18: FTCS Scheme -Curve for ns = 21

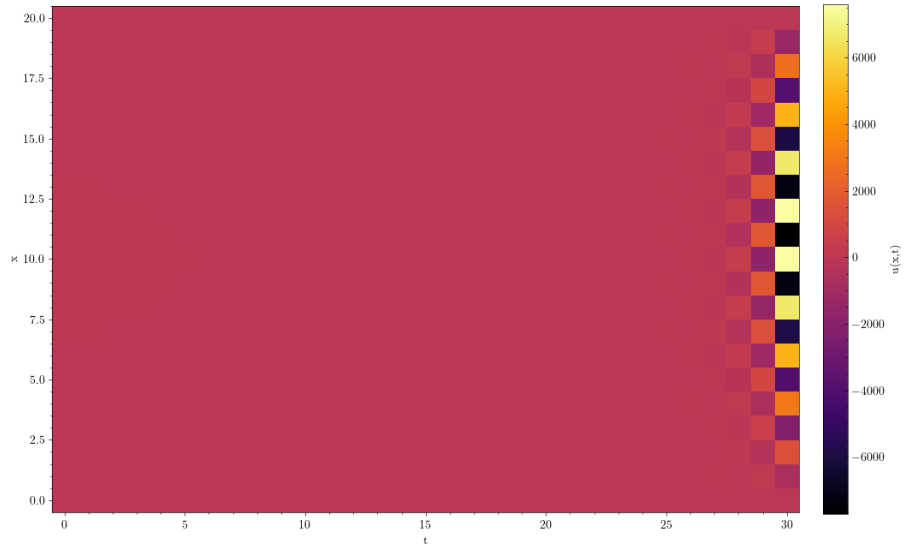


Figure 19: FTCS Scheme - Heatmap for ns = 21

Using the same code as the previous section, only change ns = 21, we can get the result in figure 18 and 19. The numerical solution gets unstable because now $\alpha = 1.333 \geq 0.5$. The value of FTCS schemes fluctuate and die out extremely fast at the beginning of the simulation.

8 Explicit Schemes for Advection equation

8.1 Advection equation

The advection equation can be expressed as:

$$\frac{\partial C}{\partial t} + u\frac{\partial C}{\partial x} = 0$$

Consider spatial discretization: x = m Δx, temporal discretization: t = n Δt, adopt the notation that C(x,t) = C(mΔx, nΔt) = $C_{m,n}$.

Courant–Friedrichs–Lewy (CFL) criterion

The convergence condition by Courant–Friedrichs–Lewy is a necessary condition for convergence while solving certain partial differential equations numerically. This criterion is the constraint between the spatial and temporal interval in solving the advection equation. In general, for a stable numerical solution, as the temporal interval gets smaller, the time interval also gets smaller and vice versa.

Centered in time, centered in space (CTCS)

$$\frac{C_{m,n+1} - C_{m,n-1}}{2\Delta t} + u\frac{C_{m+1,n} - C_{m-1,n}}{2\Delta x} = 0$$
$$\implies \quad C_{m,n+1} = C_{m,n-1} - \frac{u\Delta t}{\Delta x}(C_{m+1,n} - C_{m-1,n})$$

The CFL condition for CTCS scheme is that $|\frac{u\Delta t}{\Delta x}| \leq 1$

Forward in time, upstream in space (FTUS)

$$\frac{C_{m,n+1} - C_{m,n}}{\Delta t} + u\frac{C_{m,n} - C_{m-1,n}}{\Delta x} = 0$$
$$\implies \quad C_{m,n+1} = C_{m,n} - \frac{u\Delta t}{\Delta x}(C_{m,n} - C_{m-1,n})$$

The CFL condition for FTUS scheme is similar to CTCS scheme: $|\frac{u\Delta t}{\Delta x}| \leq 1$

Physical and numerical modes

The physical mode represents the behavior of the solution that is genuinely part of the underlying physical problem. In the case of the advection equation, the physical mode corresponds to the true advection of the quantity C with the velocity u.

The numerical mode, on the other hand, is an artifact introduced by the discretization and numerical solution method. It is not a part of the physical problem but arises due to the discrete nature of the numerical scheme. Numerical modes can lead to spurious oscillations or instability in the numerical solution.

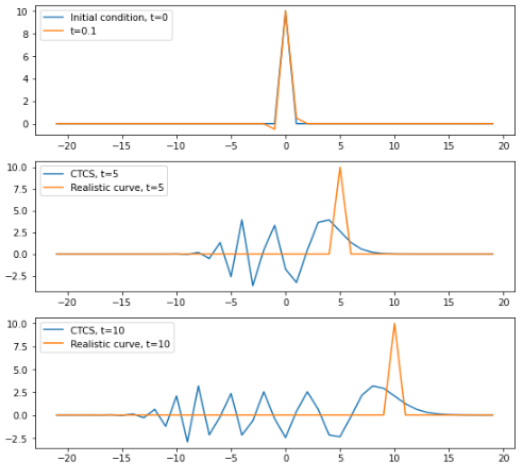
Practice #8.1

Practice #8.1 with Python

CTCS, FTCS schemes

- C(0,0)=10; C(x,0)=0 elsewhere
- u=1, Δx=1, Δt=0.1

1. **Plot the CTCS solution for different t values (e.g. Figure) with FTCS at Δt**
2. Overlay the realistic curve
3. Overlay the FTCS solution for all time steps



Hint.

$$C_{m,n+1} = C_{m,n-1} - \frac{u\Delta t}{\Delta x}\left(C_{m+1,n} - C_{m-1,n}\right)$$

$$C_{m,1} = C_{m,0} - \frac{u\Delta t}{2\Delta x}\left(C_{m+1,0} - C_{m-1,0}\right)$$

15

```python
x = np.arange(-20,20)
ini = np.zeros_like(x)
id_center = np.where(x == 0)[0][0]
ini[id_center] = 10

u = 1
dx = 1
dt = 0.1
alpha = u*dt/dx
t_max = 10
steps = int(t_max/dt)

CTCS = np.zeros((steps,len(x)))
CTCS[0,:] = ini

def C_real(t):
    C_real = np.zeros_like(ini)
    idx = np.where(x == (x[id_center] + u*t))[0][0]
    step = int(t/dt) - 1
    C_real[idx] = 10

    return C_real, step
```

```
23
24 for i in range(1, len(ini) - 1):
25     CTCS[1,i] = ini[i] - alpha/2 * (ini[i+1] - ini[i-1])
26
27 for j in range(1,steps - 1):
28     for i in range(1,len(ini)-1):
29         CTCS[j+1,i] = CTCS[j-1,i] - alpha*(CTCS[j,i+1] - CTCS[j,i-1])
30
31 fig, (ax1, ax2, ax3) = plt.subplots(3,1, sharex=True, figsize=(10,6))
32
33 color1 = 'red'
34 color2 = 'blue'
35
36 ax1.plot(x,CTCS[0],color = color1,label='initial condition, t = 0')
37 ax1.plot(x,CTCS[1,:], color = color2,label='FTCS, t = 0.1')
38 ax1.legend()
39
40 t1 = 5
41 C_real_plot1, c_step1 = C_real(t1)
42 ax2.plot(x, C_real_plot1, color=color1, label = f'Realistic curve, t = {t1}'
       )
43 ax2.plot(x, CTCS[c_step1,:],color = color2, label = f'CTCS, t = {t1}')
44 ax2.legend()
45
46 t2 = 10
47 C_real_plot2, c_step2 = C_real(t2)
48 ax3.plot(x, C_real_plot2, color = color1, label = f'Realistic curve, t = {t2
     }')
49 ax3.plot(x, CTCS[c_step2,:], color = color2, label = f'CTCS, t = {t2}')
50 ax3.legend()
51
52 plt.tight_layout()
53 plt.show()
```

In this problem, we use FTCS scheme at the beginning t = Δt = 0.1, because the CTCS scheme can not approximate the first time step. The following steps are calculated using CTCS schemes with cfl condition equal to 0.1, so this solution is expected to be stable. The results are at t = 0.1, t = 5 and t = 10 are given in figure 20. The first peak from the right of blue curve is the physical mode, we can see that in this case, the physical mode poorly approximate the position and value of the realistic curve. At the same time, the numerical solution introduced many numerical modes, with the amplitude somewhat similar to the physical mode. Therefore, despite being a stable solution, this is not a good scheme for the advection of this particular curve.

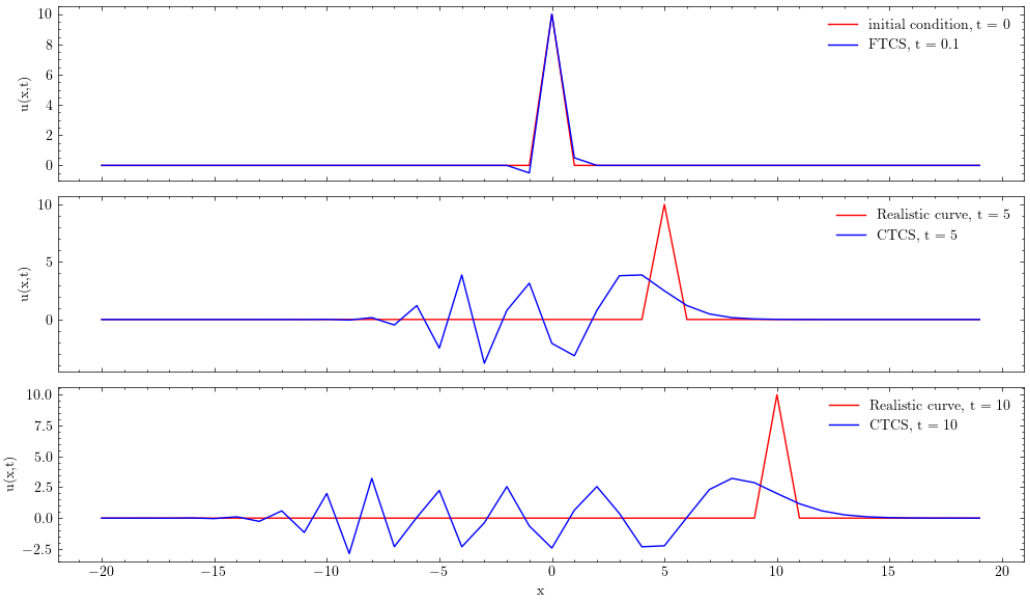
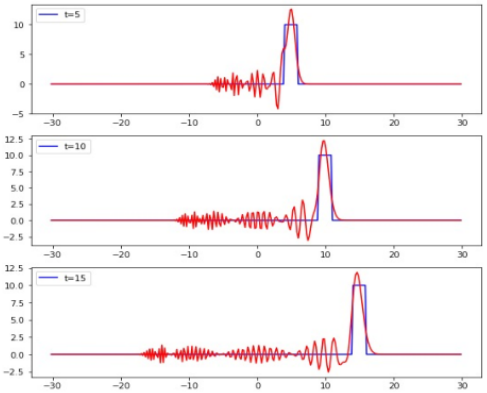


Figure 20: CTCS, FTCS schemes

Practice #8.2

Practice #8.2 with Python

CTCS, FTCS, CFL criterion



advection of a rectangular profile (thin blue curve): Δx=0.2, Δt=0.1, u=1.
Initial condition: C=10 for -1≤x≤1 & C=0 else.

1.Reproduce the above figure
- plot the blue curves
- overlay the CTCS solutions (red curves)

2.Overlay the FTCS results
3. Overlay the FTUS results
4.Plot the figure for Δt=0.5 (? remark on the CFL criterion)

16

```
u = 1
dx = 0.2
dt = 0.1
alpha = u*dt/dx

x_max = 30
t_max = 20

x = np.arange(-x_max,x_max,dx)
id_center = int(len(x)/2)
steps = int(t_max/dt)

ini = np.zeros(len(x))

x_square = [-1,1]
id_x_square = np.where((x_square[0] <= x) & (x <= x_square[1]))[0]
ini[id_x_square] = 10

epsilon = dx/2
def C_real(t):
    C_real = np.zeros_like(ini)
    idx = np.where((x <= (x[id_center] + u*t + epsilon)) & (x >= (x[
   id_center] + u*t - epsilon)))[0][0]
    step = int(t/dt) - 1
    C_real[idx-5:idx+5] = 10
    return C_real, step

# CTCS schemes
CTCS = np.zeros((steps,len(x)))
CTCS[0,:] = ini

for i in range(1, len(ini) - 1):
    CTCS[1,i] = ini[i] - alpha/2 * (ini[i+1] - ini[i-1])

for j in range(1,steps - 1):
    for i in range(1,len(ini)-1):
        CTCS[j+1,i] = CTCS[j-1,i] - alpha*(CTCS[j,i+1] - CTCS[j,i-1])

# FTUS schemes
FTUS = np.zeros((steps,len(x)))
FTUS[0,:] = ini

for j in range(0,steps - 1):
    for i in range(1,len(ini)):
        FTUS[j+1,i] = FTUS[j,i] - alpha*(FTUS[j,i] - FTUS[j,i-1])

def FTCS_schemes(dt = 0.01):
    u = 1
```

```
    dx = 0.2
    dt = 0.01
    alpha = u*dt/dx

    steps = int(t_max/dt)

    x_square = [-1,1]
    id_x_square = np.where((x_square[0] <= x) & (x <= x_square[1]))[0]

    # FTCS schemes
    FTCS = np.zeros((steps,len(x)))
    FTCS[0,:] = ini
    for j in range(steps - 1):
        for i in range(1, len(ini) - 1):
            FTCS[j+1,i] = FTCS[j,i] - alpha/2*(FTCS[j,i+1] - FTCS[j,i-1])
    return FTCS

fig, ax = plt.subplots(3,1, sharex=True, figsize=(12,6))

color = ['black','dodgerblue','red','limegreen']

t = [5,10,15]

def plot(i,t):
    C_real_plot1, c_step1 = C_real(t)
    ftcs_step = int(t/0.01) - 1
    ax[i].plot(x,C_real_plot1, color=color[0], label = f'Realistic Curve, t
   = {t}')
    ax[i].plot(x,ftcs[ftcs_step,:],color = color[3], label = f'FTCS, t={t}')
    ax[i].plot(x,CTCS[c_step1,:],color = color[1], label = f'CTCS, t={t}')
    ax[i].plot(x,FTUS[c_step1,:],color = color[2], label = f'FTUS, t={t}')
    ax[i].legend()

for i in range(len(t)):
    plot(i, t[i])

plt.tight_layout()
plt.savefig("figures/exercise_figure/ex8_prac2")
```

The CFL criterion for this problem is 0.5 for CTCS and FTUS and 0.05 for FTCS, I have to decrease the value of Δt to 0.01 because FTCS scheme is not a stable scheme like CTCS and FTUS. From figure 22, the FTCS scheme (green curve) is extremely unstable and produce huge numerical modes, and also computationally expensive. The CTCS scheme is good for approximate the rectangular profile, because the physical mode is at the right position and decent amplitude. Because of the centered differences, the numerical modes still appear, but

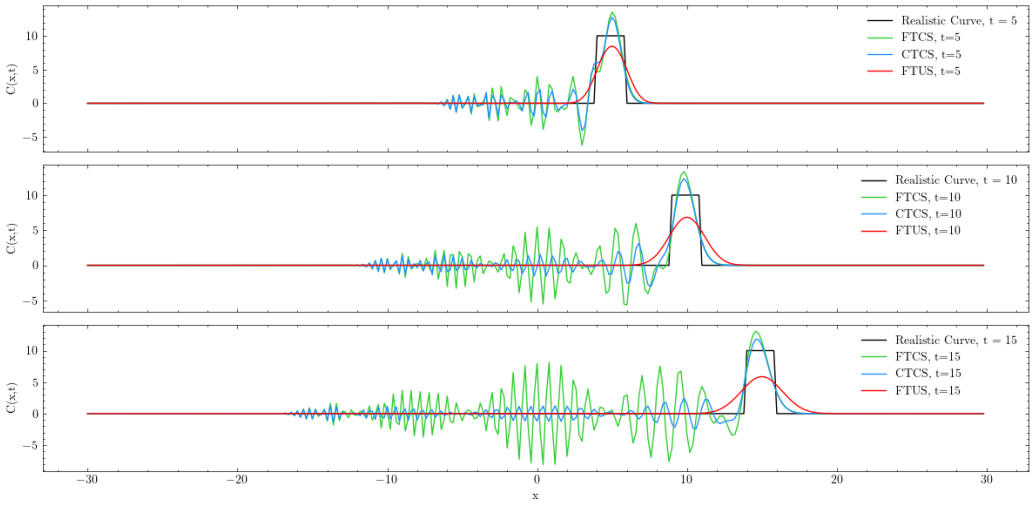


Figure 21: CTCS, FTCS, FTUS schemes

they are small compared to physical modes. The FTUS scheme, on the other hand, do not product numerical modes, but is affected by a very strong damping and dispersion.

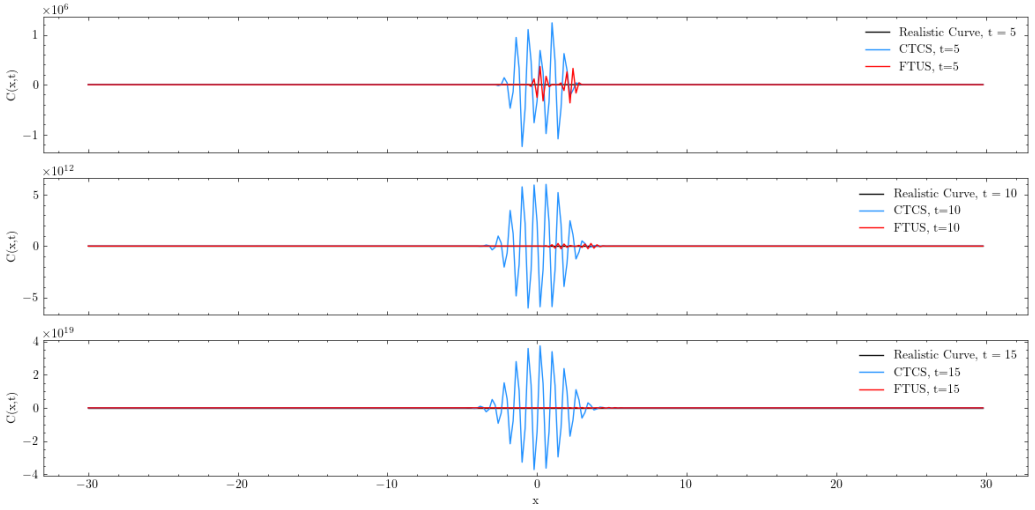


Figure 22: CTCS, FTUS schemes, case of dt = 0.5

When dt = 0.5, the CFL condition is not satisfied, as CFL equal to 2.5 > 1, so the CTCS scheme get unstable and explode, while FTUS get damped and die out very quickly.

9 Implicit Schemes for Advection equation

9.1 Numerical methods of Implicit schemes for Advection equation

Implicit trapezoidal scheme

$$\frac{C_{m,n+1} - C_{m,n}}{\Delta t} + u\frac{1}{2}(\frac{C_{m+1,n+1} - C_{m-1,n+1}}{2\Delta x} + \frac{C_{m+1,n} - C_{m-1,n}}{2\Delta x}) = 0$$

The numerical solution are be calculated using matrix, the $\vdots$ and $\dots$ are 0:

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & -\frac{u\Delta t}{4\Delta x} & -1 & \frac{u\Delta t}{4\Delta x} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ C_{m-1,n} \\ C_{m,n} \\ C_{m+1,n} \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & -\frac{u\Delta t}{4\Delta x} & 1 & \frac{u\Delta t}{4\Delta x} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ C_{m-1,n+1} \\ C_{m,n+1} \\ C_{m+1,n+1} \\ \vdots \end{bmatrix} = 0$$

We can rewrite as

$$\begin{aligned} A\vec{C_n} + B\vec{C_{n+1}} &= 0 \\ \implies \quad \vec{C_{n+1}} &= -B^{-1}A\vec{C_n} \end{aligned}$$

Implicit leap frog scheme

$$\frac{C_{m,n+1} - C_{m,n}}{\Delta t} + u(\frac{C_{m+1,n+1} - C_{m-1,n+1}}{2\Delta x}) = 0$$

The numerical solution are be calculated using matrix, the $\vdots$ and $\dots$ are 0:

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & 0 & -1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ C_{m-1,n} \\ C_{m,n} \\ C_{m+1,n} \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & -\frac{u\Delta t}{2\Delta x} & 1 & \frac{u\Delta t}{2\Delta x} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ C_{m-1,n+1} \\ C_{m,n+1} \\ C_{m+1,n+1} \\ \vdots \end{bmatrix} = 0$$

We can rewrite as

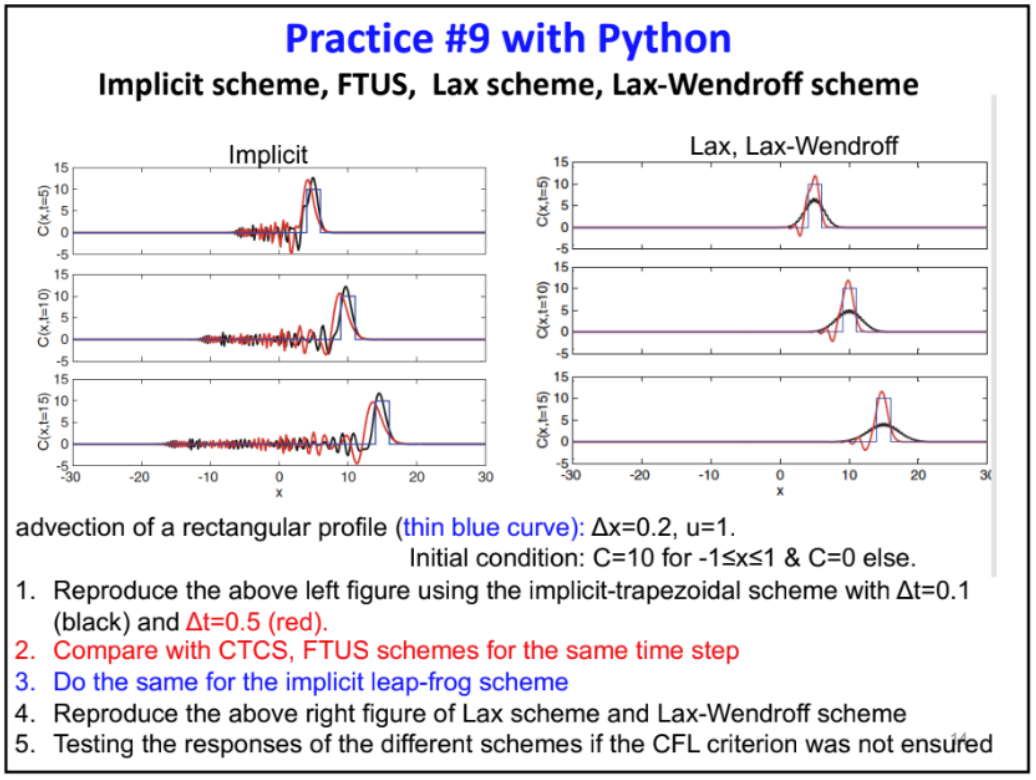
$$\begin{aligned} E\vec{C_n} + F\vec{C_{n+1}} &= 0 \\ \implies \quad \vec{C_{n+1}} &= -F^{-1}E\vec{C_n} \end{aligned}$$

Implicit Lax scheme

$$C_{m,n+1} = C_{m,n} - \frac{u\Delta t}{2\Delta x}(C_{m+1,n} - C_{m-1,n}) + \frac{1}{2}(C_{m+1,n} + C_{m-1,n} - 2C_{m,n})$$

Implicit Lax Wandroff scheme

$$C_{m,n+1} = C_{m,n} - \frac{u\Delta t}{2\Delta x}(C_{m+1,n} - C_{m-1,n}) + \frac{u^2\Delta t^2}{2\Delta x^2}(C_{m+1,n} + C_{m-1,n} - 2C_{m,n})$$



9.2 Implicit Trapezoidal Scheme

```
u = 1
dx = 0.2
dt = 0.1
alpha = u*dt/dx

x_max = 30
t_max = 20

x = np.arange(-x_max,x_max,dx)
id_center = int(len(x)/2)
steps = int(t_max/dt)

ini = np.zeros(len(x))

x_square = [-1,1]
id_x_square = np.where((x_square[0] <= x) & (x <= x_square[1]))[0]
ini[id_x_square] = 10

epsilon = dx/2
def C_real(t):
```

```
    C_real = np.zeros(len(x))
    idx = np.where((x <= (x[id_center] + u*t + epsilon)) & (x >= (x[
   id_center] + u*t - epsilon)))[0][0]
    step = int(t/dt) - 1
    C_real[idx-5:idx+5] = 10
    return C_real, step

# CTCS schemes
CTCS = np.zeros((steps,len(x)))
CTCS[0,:] = ini

for i in range(1, len(ini) - 1):
    CTCS[1,i] = ini[i] - alpha/2 * (ini[i+1] - ini[i-1])

for j in range(1,steps - 1):
    for i in range(1,len(ini)-1):
        CTCS[j+1,i] = CTCS[j-1,i] - alpha*(CTCS[j,i+1] - CTCS[j,i-1])

# FTUS schemes
FTUS = np.zeros((steps,len(x)))
FTUS[0,:] = ini

for j in range(0,steps - 1):
    for i in range(1,len(ini)):
        FTUS[j+1,i] = FTUS[j,i] - alpha*(FTUS[j,i] - FTUS[j,i-1])

# Implicit trapezoidal schemes
A = np.zeros((len(x), len(x)))
B = np.zeros((len(x), len(x)))

for m in range(0,len(x)):
    A[m,m] = -1
    B[m,m] = 1

for m in range(1, len(x) - 1):
    A[m,m-1] = -alpha/4
    A[m,m+1] = alpha/4
    B[m,m-1] = -alpha/4
    B[m,m+1] = alpha/4

A = np.matrix(A)
B = np.matrix(B)
B_inv = B.I

ini_im = ini
implicit = np.zeros((steps,len(x)))
implicit[0,:] = ini_im
for i in range(1,steps):
    ini_im = np.matrix(implicit[i-1,:])
```

```
    next_cell = np.zeros_like(ini_im)
    next_cell = -B_inv*A*ini_im.T
    implicit[i,:] = np.array(next_cell.T)[0]

fig, ax = plt.subplots(3,1, sharex=True, figsize=(12,6))

color = ['black','dodgerblue','red','limegreen','violet','gold','hotpink']

t = [5,10,15]

def plot(i,t):
    C_real_plot1, c_step1 = C_real(t)
    ax[i].plot(x,C_real_plot1, color=color[0], label = f'Realistic Curve, t = {t}')
    ax[i].plot(x,CTCS[c_step1,:],color = color[1], label = f'CTCS, t={t}')
    ax[i].plot(x,FTUS[c_step1,:],color = color[3], label = f'FTUS, t={t}')
    ax[i].plot(x,implicit[c_step1,:],color = color[2], linestyle='-', linewidth=2, label = f'Implicit trapezoidal, t={t}')
    ax[i].legend()

for i in range(len(t)):
    plot(i, t[i])

plt.tight_layout()
plt.show()
```

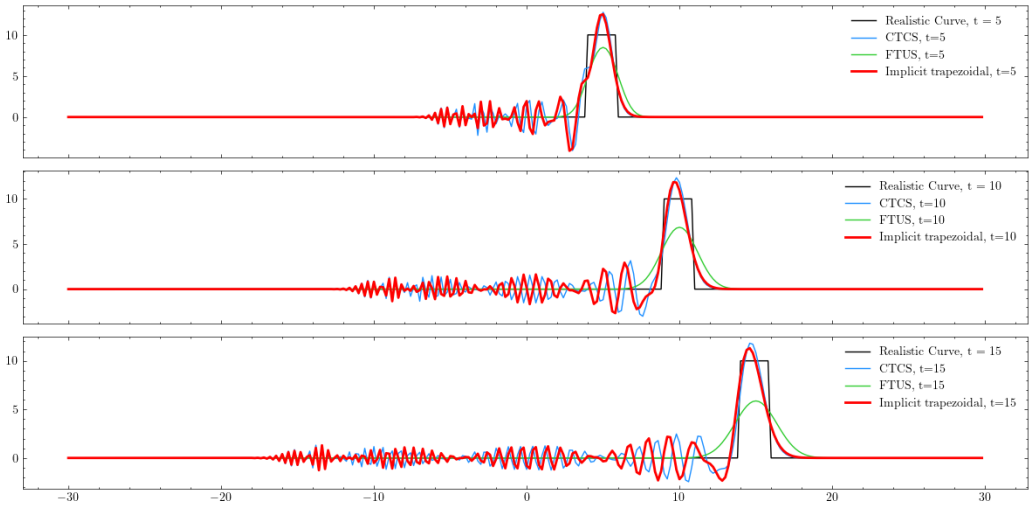


Figure 23: Implicit Trapezoidal Scheme

The CFL in this problem is 0.5, a stable value for the implicit trapezoidal scheme. From

the figure 23, we can compare the implicit trapezoidal scheme with explicit CTCS and FTUS scheme in the previous section. We can see that, the implicit trapezoidal scheme displays a same behavior to the explicit CTCS scheme with a high physical modes and many small numerial modes.

9.3 Implicit Leap Frog Scheme

```
# Implicit leap frog
A = np.zeros((len(x), len(x)))
B = np.zeros((len(x), len(x)))

for m in range(0,len(x)):
    A[m,m] = -1
    B[m,m] = 1

for m in range(1, len(x) - 1):
    A[m,m-1] = -alpha/4
    A[m,m+1] = alpha/4

A = np.matrix(A)
B = np.matrix(B)
B_inv = B.I

ini_im = ini
implicit_lf = np.zeros((steps,len(x)))
implicit_lf[0,:] = ini_im
for i in range(1,steps):
    ini_im = np.matrix(implicit_lf[i-1,:])
    next_cell = np.zeros_like(ini_im)
    next_cell = -B_inv*A*ini_im.T
    implicit_lf[i,:] = np.array(next_cell.T)[0]
```

The implicit leap frog scheme do not converge, so it is unstable, with the given dt and dx, the solutionn exploded as we can see from figure 24.

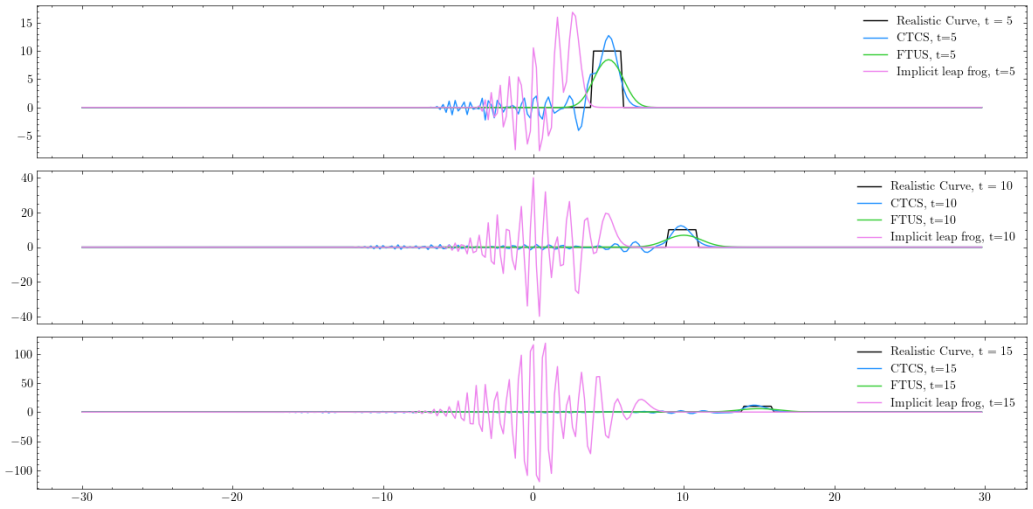


Figure 24: Implicit Leap Frog Scheme

9.4 Implicit Lax and Lax Wandroff Scheme

```
# Lax scheme
Lax = np.zeros((steps,len(x)))
Lax[0,:] = ini

for j in range(0,steps - 1):
    for i in range(1,len(ini)-1):
        Lax[j+1,i] = Lax[j,i] - alpha/2*(Lax[j,i+1] - Lax[j,i-1]) + 1/2 * (
   Lax[j,i+1] - 2*Lax[j,i] + Lax[j,i-1])

# Lax-Wendroff scheme
Lax_W = np.zeros((steps,len(x)))
Lax_W[0,:] = ini

for j in range(0,steps - 1):
    for i in range(1,len(ini)-1):
        Lax_W[j+1,i] = Lax_W[j,i] - alpha/2*(Lax_W[j,i+1] - Lax_W[j,i-1]) +
   1/2 * alpha**2 * (Lax_W[j,i+1] - 2*Lax_W[j,i] + Lax_W[j,i-1])
```

From the figure 25, we can see that Lax scheme (black curve) does not present the numerical modes, but a strong damping. In order to prevent that, we can add a square of CFL criterion to the diffusion term (second term in Lax scheme), this is so-called Lax Wendroff scheme. The Lax Wendroff scheme can evolve without damping, but the trade-off is that there is a very small numerical modes exist. However, comparing to other schemes, the Lax Wendroff is a good balance between eliminating numerical modes and conserving physical mode's amplitude.

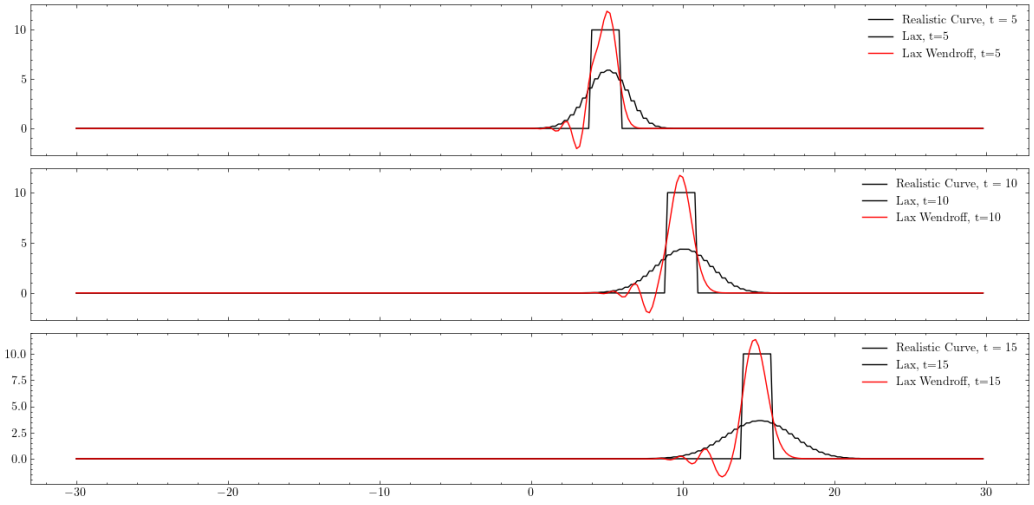


Figure 25: Lax and Lax Wandroff Scheme

10 Chaos Theory

Chaos theory explores the behavior of dynamic systems (anything evolves with time) that are highly sensitive to initial conditions. In the context of numerical simulations, even slight variations in the starting conditions can lead to vastly different outcomes over time. This sensitivity, often characterized by the butterfly effect, is a fundamental aspect of chaotic systems.

This butterfly effect is the combination of *nonlinearity* and *dissipation*. Nonlinearity is the system that are not linear. Dissipation is the system that has some friction (e.g viscosity), as t $\longrightarrow \infty$, system approaches an *attractor* which does not depend (usually) on initial conditions.

Almost all natural systems are nonlinear and dissipative. Climate is also part of them, as it involves fluids, convection and life for example, this explains for plenty of examples of abrupt climate change in the past.

Lorenz Attractor

The Lorenz Attractor is a mathematical model that exhibits chaotic behavior. It was introduced by Edward Lorenz in 1963 as a simplified atmospheric convection model. The equations governing the Lorenz Attractor are a system of three ordinary differential equations:

$$\begin{aligned}\frac{dx}{dt} &= P(y - x),\\ \frac{dy}{dt} &= x(R - z) - y,\\ \frac{dz}{dt} &= xy - Bz,\end{aligned}$$

where x, y, and z are variables representing the state of the system, and P, R and B are parameters.

Verhulst model

The Verhulst model describes how a population grows in an environment with limited resources. It says that as a population gets closer to the maximum number it can sustain, the growth slows down. Verhulst also contain bifurcation behavior or switching behavior when the parameter or resource is varied. Imagine you're gradually changing something in the environment, let's say the available space or food for the population. As you change this factor, you might notice a switch in how the population behaves.

In the Verhulst model, this change could mean that, for certain values of resources, the population might grow steadily and level off, but for other values, it might start showing unpredictable or chaotic behavior. This switch in behavior is what we call bifurcation.

10.1 Pertubation

- The prognostic variable "V" is calculated with time step of "n" following the below equation:

$$V(n+1) = 21/8 \times V(n) - 8/8 \times V^3(n)$$

where n = 0,1,2,...

- In the control calculation, V(0) = 0.1.
- Perturb the initial condition V(0) and examine the difference in time between the control and the perturbation runs.

```
def V_func(n, V_ini):
    V = np.zeros(n)
    V[0] = V_ini

    for i in range(1,n):
        V[i] = 21/8 * V[i-1] - 28/8 * V[i-1]**3
    return V

fig = plt.figure(figsize=(10,6))
n = 100
V_ini = 0.1
V_ini_1 = 0.10001

color = ['red','blue','gold']
plt.plot(np.arange(n), V_func(n, V_ini), color = color[0], label = f'V
    [0] = {V_ini}')
plt.plot(np.arange(n), V_func(n, V_ini_1), color = color[1], label = f'V
    [0] = {V_ini_1}')
plt.plot(np.arange(n), V_func(n, V_ini) - V_func(n, V_ini_1), color=
    color[2], label='Difference')

plt.legend()
plt.show()
```

The given equation is a nonlinear equation with a first degree and third degree polynomial terms. From figure 26, we see that changing just 0.00001 in the initial condition leading to an huge difference in the V(x) value that does not present any observable pattern.

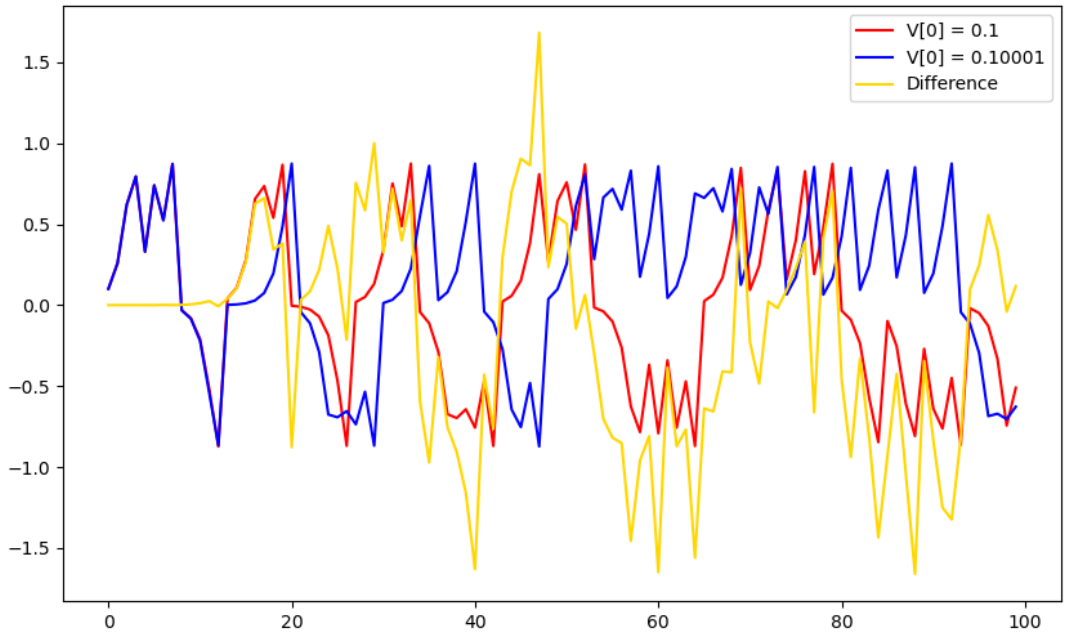


Figure 26: Pertubation

10.2 Lorenz Attractor

- Write a python code to solve the Lorenz equations

$$dx/dt = P(y - x)$$
$$dy/dt = Rx - y - xz$$
$$dz/dt = xy - Bz$$

  where P=10, R=28, B=8/3

- x[0]=1.; y[0]=1.; z[0] = 1.
- dt=0.01
- Compute for 10000 time steps

1. Re-plot the figures shown in previous slides with Euler forward in time:
   - in 3D, in xy, xz, yz plans;
   - Slightly change the initial conditions $\longrightarrow$ plot the "difference" figure

2. (Optional) How about using the centered difference scheme?
3. (Optional) How about the Runge-Kutta method?

```
def lorenz_attractor(xyz, p=10,r=28,b=8/3):
    x,y,z = xyz
    x_ = p*(y - x)
    y_ = r*x - y - x*z
    z_ = x*y - b*z
    return np.array([x_,y_,z_])

dt = 0.01
steps = 10000

def forward_scheme(ini, steps = steps, dt = dt):
    xyz = np.zeros((steps, 3))
    xyz[0] = ini
    for i in range(steps - 1):
        xyz[i+1] = xyz[i] + lorenz_attractor(xyz[i])*dt
    return xyz

ax = plt.figure(figsize=(15,15)).add_subplot(projection='3d')
ax.plot(*forward_scheme([1,1,1]).T,color='black',lw=0.7, label='initial at
    (1,1,1)')
ax.plot(*forward_scheme([1.1,1.1,1.1]).T,linestyle = '--',color='red',lw
    =0.7, label='initial at (1.1,1.1,1.1)')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")
ax.legend()
ax.set_title("Lorenz Attractor - Forward Scheme")
plt.savefig("figures/exercise_figure/ex10_lorenz_attractor_forward_.png")
```

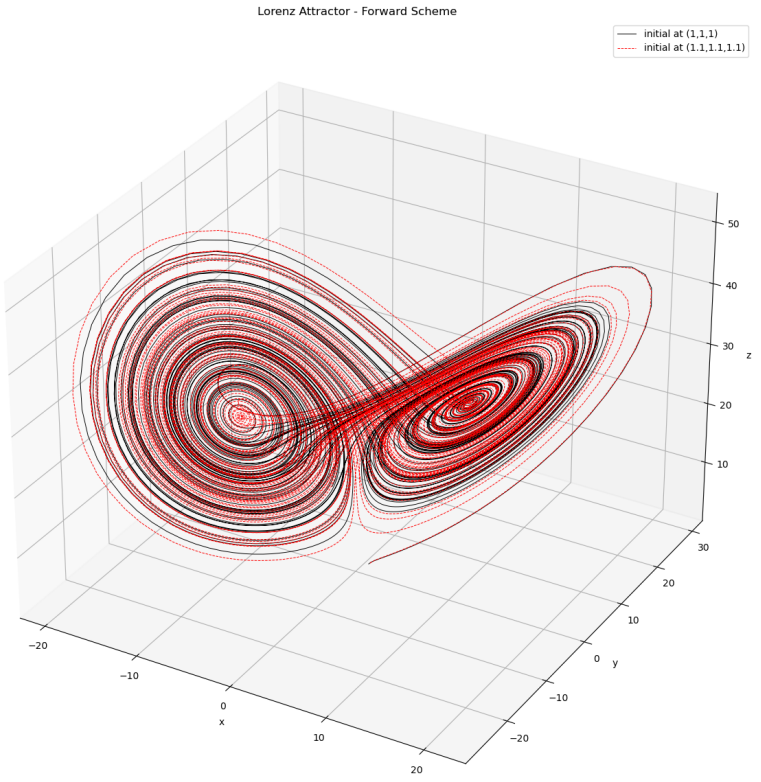


Figure 27: Lorenz Attractor with dt = 0.01, 10000 steps by forward scheme compare with slight change in initial condition

```
colors = ['red', 'green', 'blue']

fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20,4))
ax1.plot(xyz[:, 0], xyz[:, 1], color=colors[0], linewidth=0.1)
ax1.set_xlabel("x")
ax1.set_ylabel("y")
ax1.set_title("xy plane")

ax2.plot(xyz[:, 0], xyz[:, 2], color=colors[1], linewidth=0.1)
ax2.set_xlabel("x")
ax2.set_ylabel("z")
```

```
ax2.set_title("xz plane")
ax = plt.figure(figsize=(15,15)).add_subplot(projection='3d')
ax.plot(*xyz.T,lw=0.7,color='black',label = "forward")
ax.plot(*xyz1.T,lw=0.7,color='gold', label = "centered")

ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")

ax.set_title("Lorenz Attractor - Forward/Centered Scheme")
plt.savefig("figures/exercise_figure/ex10_lorenz_attractor_centered")
ax3.plot(xyz[:, 1], xyz[:, 2], color=colors[2], linewidth=0.1)
ax3.set_xlabel("y")
ax3.set_ylabel("z")
ax3.set_title("yz plane")

plt.savefig("figures/exercise_figure/ex10_lorenz_attractor_projected")
```

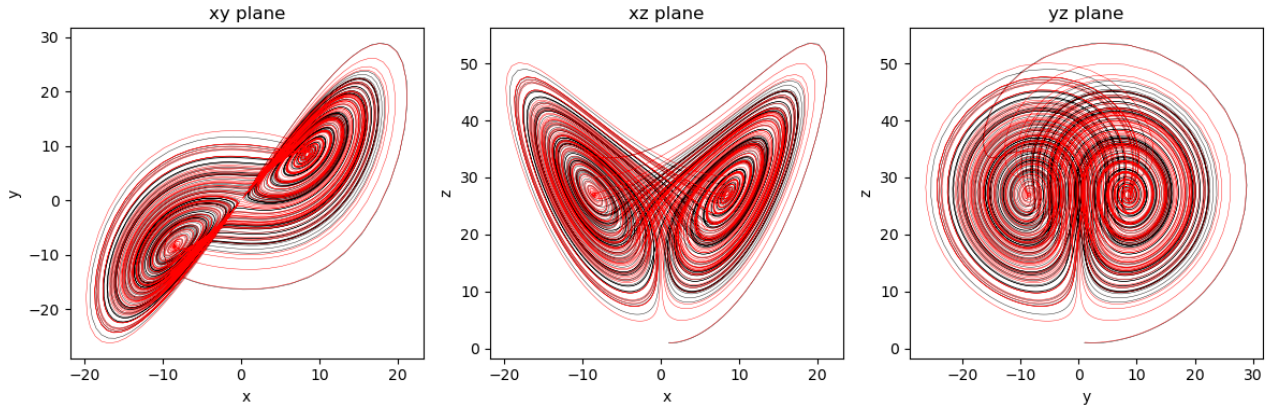


Figure 28: Lorenz Attractor projected on xy, yz, zx plane

The Lorenz Attractor in figure 27 and 30 display the nonlinear, non-periodic and deterministic butterfly-like trajectories in 3D phase space. Small changes in initial conditions (red curve) can lead to quite different trajectory, displaying the sensitive dependence on initial conditions, feature of chaotic systems.

Centered Scheme

```
def central_scheme(xyz, dt, lorenz_attractor):
    half_step = lorenz_attractor(xyz + 0.5 * dt * lorenz_attractor(xyz))
    xyz_new = xyz + dt * half_step
    return xyz_new
```

```
xyz1 = np.ones_like(xyz)
for i in range(steps - 1):
    xyz1[i+1] = central_scheme(xyz1[i], dt, lorenz_attractor)

ax = plt.figure(figsize=(15,15)).add_subplot(projection='3d')
ax.plot(*xyz.T,lw=0.7,color='black',label = "forward")
ax.plot(*xyz1.T,lw=0.7,color='gold', label = "centered")

ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")

ax.set_title("Lorenz Attractor - Forward/Centered Scheme")
plt.savefig("figures/exercise_figure/ex10_lorenz_attractor_centered")
```

Changing to other scheme, also affect the trajectory of Lorenz Attractor. Using the same time steps, the forward scheme approaches the attractors faster than the centered scheme.

Runge Kutta

```
# Parameters
p, r, b = 10, 28, 8/3
dt = 0.01
steps = 10000

# Initial conditions
xyz = np.zeros((steps, 3))
xyz[0] = [1, 1, 1]

# Slightly different initial conditions
xyz_ = np.zeros((steps, 3))
xyz_[0] = [1.0001, 1.0001, 1.0001]

# Lorenz equations
def lorenz_equations(x, y, z):
    dx = p * (y - x)
    dy = r * x - y - x * z
    dz = x * y - b * z
    return dx, dy, dz

# Runge-Kutta method
for i in range(steps-1):
    x, y, z = xyz[i]
    k1, l1, m1 = lorenz_equations(x, y, z)

```

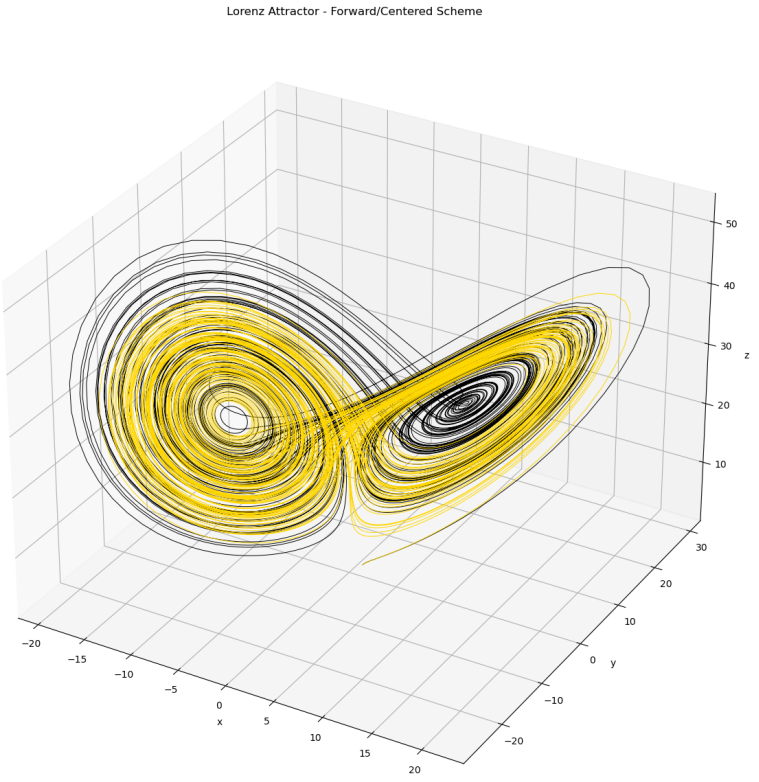


Figure 29: Lorenz Attractor in centered scheme

```
26      k2, l2, m2 = lorenz_equations(x + 0.5 * dt * k1, y + 0.5 * dt * l1, z +
        0.5 * dt * m1)
27      k3, l3, m3 = lorenz_equations(x + 0.5 * dt * k2, y + 0.5 * dt * l2, z +
        0.5 * dt * m2)
28      k4, l4, m4 = lorenz_equations(x + dt * k3, y + dt * l3, z + dt * m3)
29
30      x_ = x + dt * (k1 + 2*k2 + 2*k3 + k4) / 6
31      y_ = y + dt * (l1 + 2*l2 + 2*l3 + l4) / 6
32      z_ = z + dt * (m1 + 2*m2 + 2*m3 + m4) / 6
33
34      xyz[i+1] = [x_, y_, z_]
35
36      # Variation
```

```
    x_, y_, z_ = xyz_[i]
    k1, l1, m1 = lorenz_equations(x_, y_, z_)

    k2, l2, m2 = lorenz_equations(x_ + 0.5 * dt * k1, y_ + 0.5 * dt * l1, z_ + 0.5 * dt * m1)
    k3, l3, m3 = lorenz_equations(x_ + 0.5 * dt * k2, y_ + 0.5 * dt * l2, z_ + 0.5 * dt * m2)
    k4, l4, m4 = lorenz_equations(x_ + dt * k3, y_ + dt * l3, z_ + dt * m3)

    x_ = x_ + dt * (k1 + 2*k2 + 2*k3 + k4) / 6
    y_ = y_ + dt * (l1 + 2*l2 + 2*l3 + l4) / 6
    z_ = z_ + dt * (m1 + 2*m2 + 2*m3 + m4) / 6

    xyz_[i+1] = [x_, y_, z_]

# Plotting
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(xyz[:, 0], xyz[:, 1], xyz[:, 2], lw=0.5, color='black', label='Initial Conditions: [1, 1, 1]')
ax.plot(xyz_[:, 0], xyz_[:, 1], xyz_[:, 2], lw=0.5, color='red', label='Initial Conditions: [1.0001, 1.0001, 1.0001]')
ax.set_title('Lorenz Attractor with Slightly Different Initial Conditions')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
ax.legend()

plt.savefig("figures/exercise_figure/ex10_lorenz_runge")
```

The Runge-Kutta scheme can also produce the Lorenz Attractor and a slight change in the initial condition of the scheme also produce a different trajectory, just like the attractor of other schemes

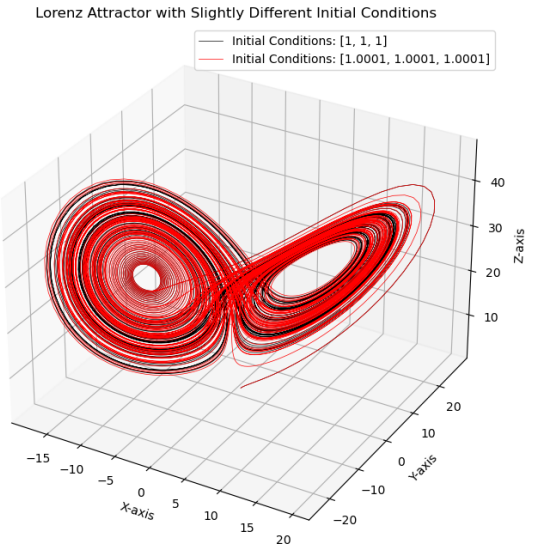


Figure 30: Lorenz Attractor in centered scheme

10.3 Verhulst Model

Write a python code to display the Bifurcation Diagrams of the Verhulst model

$$x_{n+1} = ax_n(1 - x_n)$$

```
A = np.linspace(0,4,10000)
Y = []

for a in A:
    x = np.random.random()
    for l in range(1000):
        x = a*x*(1-x)
    Y.append(x)
plt.plot(A, Y, ls='', marker=',',color = 'red')
plt.xlabel("a")
plt.ylabel("x")
plt.show()
```

In this problem, we vary the value of parameter a in range (0,4). From figure 31, we can see that, in range (0,1), the population will eventually die, independent of the initial population. In

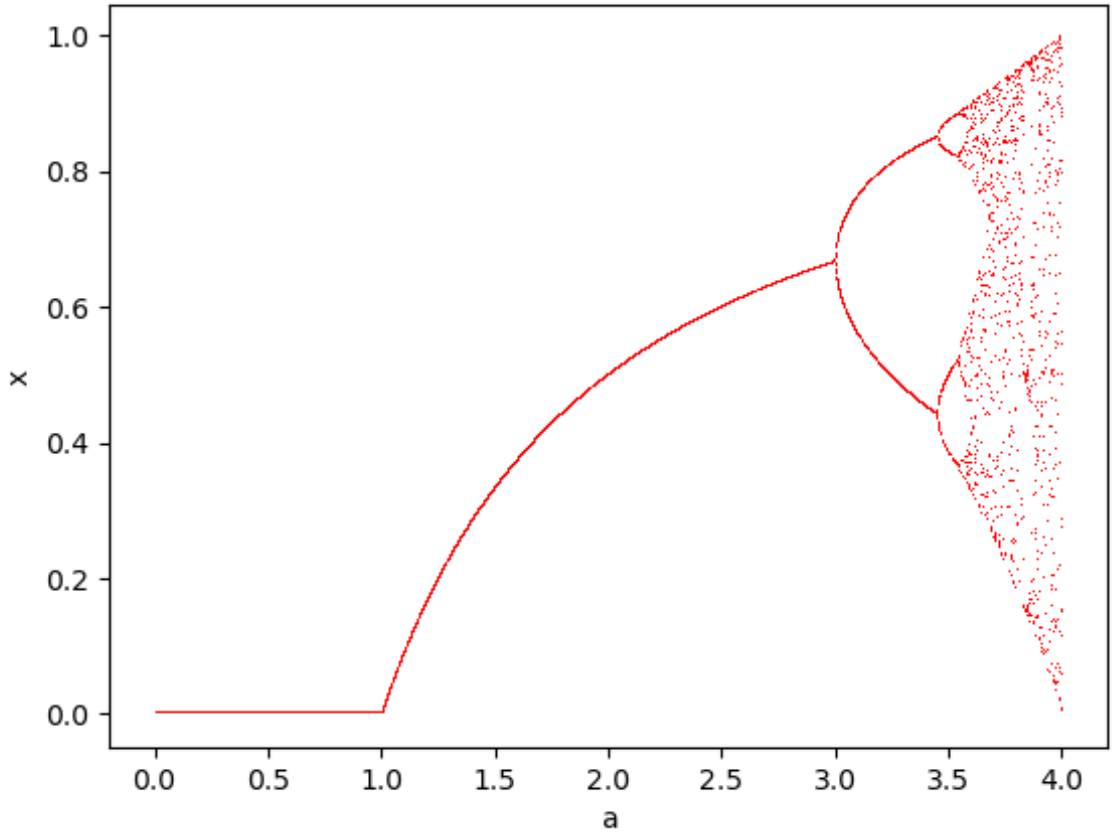


Figure 31: Verhulst Model

range (1, 3), the system will approach a single value. But from 3 to about 3.5, the population will approach an oscillations between two values or so-called bifurcation behavior. Furthermore, the population will approach an oscillations between 4, 8, 16, ... with the length of parameter interval a becomes smaller and smaller.

11 Shallow water model

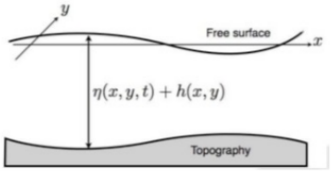
The shallow water model is a simplified approach to studying the behavior of water in large basins like oceans. It is particularly useful when the average depth of the water layer (H) is much smaller than its horizontal extent. The shallow water model makes several key assumptions:

- The ocean is treated as a homogeneous layer of water.
- The average thickness (H) is much smaller than the horizontal scale.
- Vertical accelerations ($\frac{Dw}{Dt}$) are considered small, leading to hydrostatic equilibrium in the vertical direction.

Practice #11: Shallow water equations

- Study one-dimensional gravity waves. Gravity waves can be described by the shallow-water equation

$$\frac{\partial u}{\partial t} = -g\frac{\partial \eta}{\partial x}$$
$$\frac{\partial \eta}{\partial t} = -H\frac{\partial u}{\partial x}$$



- Write a program solving the equations with the leap-frog scheme on an unstaggered grid. Set H=g=1.
- Δx=0.025, CFL-number=0.9 (=c. Δt/Δx where c=$(gH)^{1/2}$)
- Initialize the leapfrog scheme with a single Euler forward step
- Initial conditions:

$$h(x,t=0) = \begin{cases} \frac{1}{2}+\frac{1}{2}\cos\left[10\pi(x-0.5)\right], & \text{for } 0.4 \le x \le 0.6 \\ 0, & \text{elsewhere} \end{cases}$$
$$u(x,t=0) = h(x,t=0)$$

33

(from "Numerical methods in Meteorology and Oceanography")

```python
import numpy as np
import matplotlib.pyplot as plt

# Initial Condition Setup
h = lambda x: 0.5 + 0.5 * np.cos(10 * np.pi * (x - 0.5))
dx = 0.025
x = np.arange(-5, 10, dx)
n_u = len(x)
x_array = np.zeros(n_u)
 # Change the domain to -10, 10
```

```
mask = (x < 0.4) | (x > 0.6)
u_ini = h(x)
u_ini[mask] = 0

# Constants and Parameters
cfl = 0.9
g = 1
H = 1
c = (g * H) ** 0.5
dt = cfl / c * dx
t_steps = 1000
t = np.arange(0, t_steps * dt, dt)

# Arrays Initialization
u_st = np.zeros((n_u, t_steps))
u_st[:, 0] = u_ini
nu_st = u_st

# Time Stepping Loop
for i in range(1, n_u - 1):
    u_st[i, 1] = u_st[i, 0] - g * dt / (2 * dx) * (nu_st[i + 1, 0] - nu_st[i - 1, 0])
    nu_st[i, 1] = nu_st[i, 0] - H * dt / (2 * dx) * (u_st[i + 1, 0] - u_st[i - 1, 0])

for j in range(1, t_steps - 1):
    for i in range(1, n_u - 1):
        u_st[i, j + 1] = u_st[i, j - 1] - g * dt / dx * (nu_st[i + 1, j] - nu_st[i - 1, j])
        nu_st[i, j + 1] = nu_st[i, j - 1] - H * dt / dx * (u_st[i + 1, j] - u_st[i - 1, j])

# Plotting
fig, ax = plt.subplots(3, 1, sharex=True, figsize=(12, 6))

color = ['black', 'dodgerblue', 'red', 'limegreen']
t_ = [0, 0.18, 0.72]
ax[0].plot(x, u_st[:, 0], color=color[1], label=f'')
ax[1].plot(x, u_st[:, 20], color=color[1])
ax[2].plot(x, u_st[:, 200], color=color[1])

for i in range(3):
    ax[i].set_ylabel("u(x)")
plt.xlabel("x")
plt.tight_layout()
plt.savefig("figures/exercise_figure/ex11.png")
```

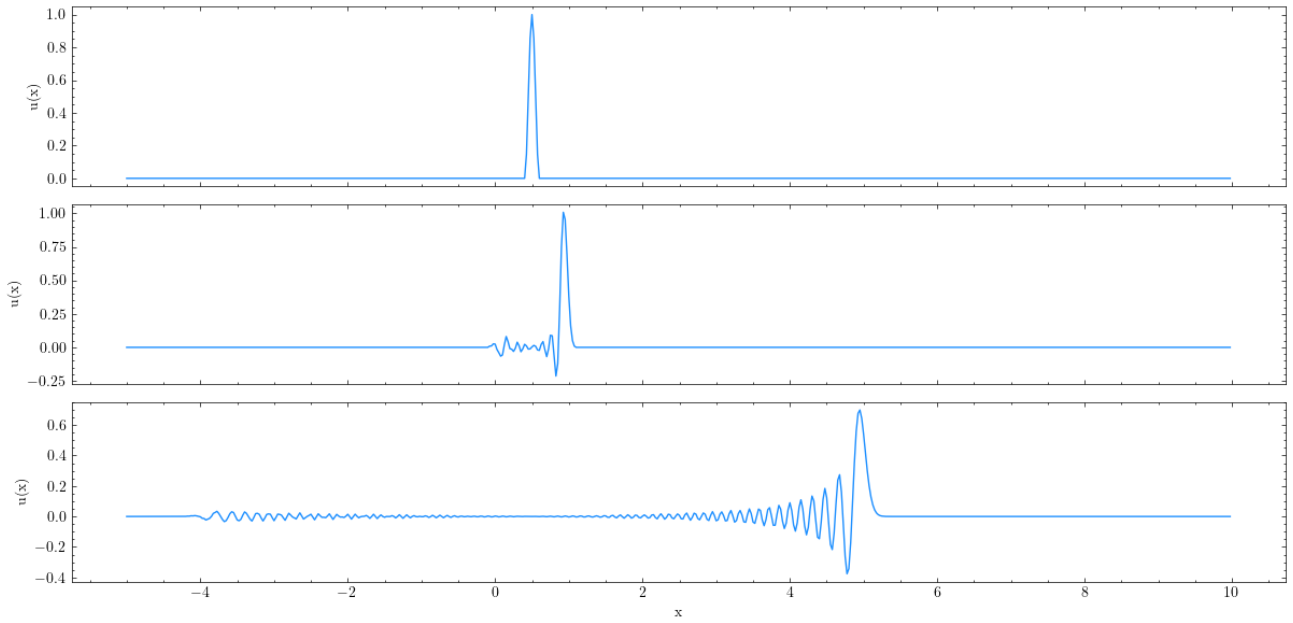


Figure 32: Shallow Water Model

The simulation of shallow water dynamics utilizes a finite difference method with a Courant-Friedrichs-Lewy (CFL) number of 0.9 to ensure numerical stability. The initial condition sets the water height (h) as a cosine function with a wavelength of 10π and an amplitude of 0.5, creating a single wave at $t = 0$. Boundary constraints are applied, setting the water height to zero for $x < 0.4$ and $x > 0.6$, introducing a discontinuity.

In Figure 32, the simulation illustrates the evolution of the water height over time. At $t = 0$, a solitary wave initiates the simulation, while at $t = 0.18$, reflections and interactions lead to changes in the wave pattern, including the emergence of smaller waves. Notably, small waves appear on the opposite side of the initial position, revealing the dynamic complexity of the shallow water system.

12 Global Climate Model (GCM)

Interactive atlas: https://interactive-atlas.ipcc.ch

1. Learn how to use the tool
2. Plot the trends of past temperature & precipitation worldwide and in Southeast Asia; and then provide comments on the results obtained
3. What should be the changes of specific variables in Southeast Asia under the 1.5oC global warming level?
4. Provide comments on the changes in rainfall and temperature at the end- century (2081–2100) compared to the baseline period under CMIP6 SSP5-8.5 and CMIP5 RCP8.5

Mean Temperature Change

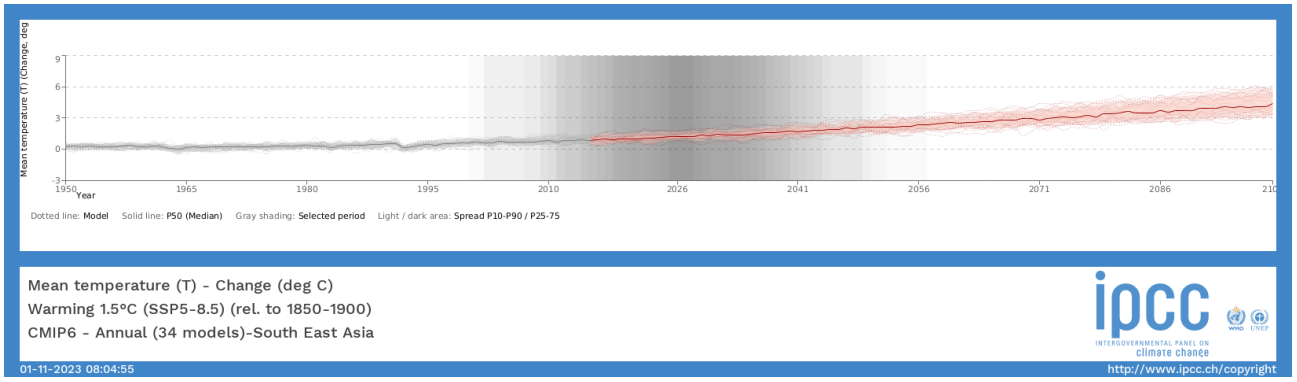


Figure 33: Mean Temperature Change of South East Asia

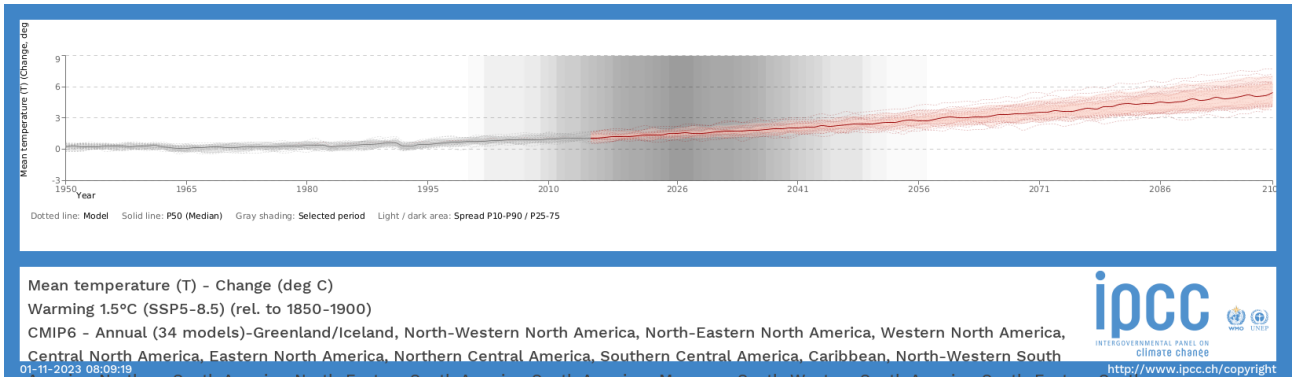


Figure 34: Mean Temperature Change of the World

We can see that the mean temperature change of South East Asia and the world from the past to present and the prediction in the future has the similar tendency of increasing. However, the increasing amount of the world is expected to be higher and higher compared to the increasing amount of South East Asia. At the end of 2100, the mean temperature change of the world is expected to be aroung 4.5^{o}C, while in South East Asia, it is expected to be 3.6^{o}C.

Total Precipitation

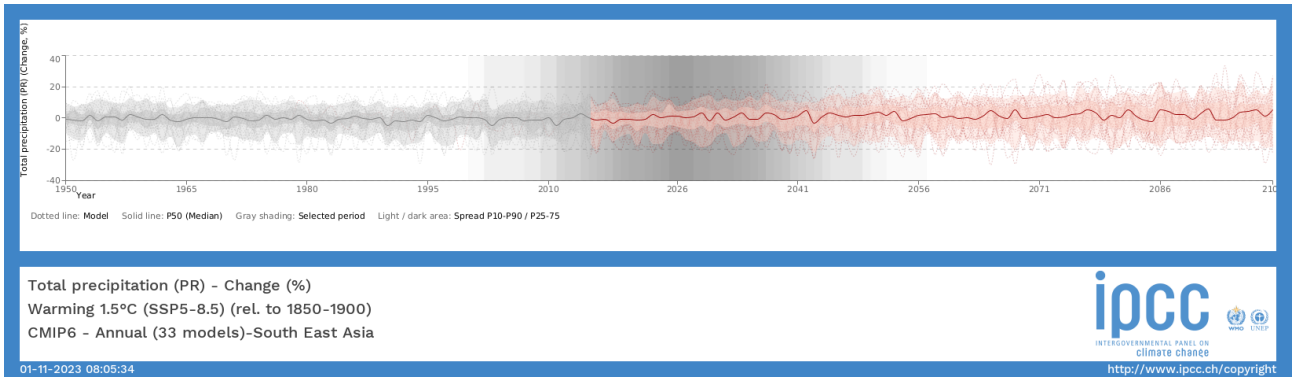


Figure 35: Total precipitation of the South East Asia

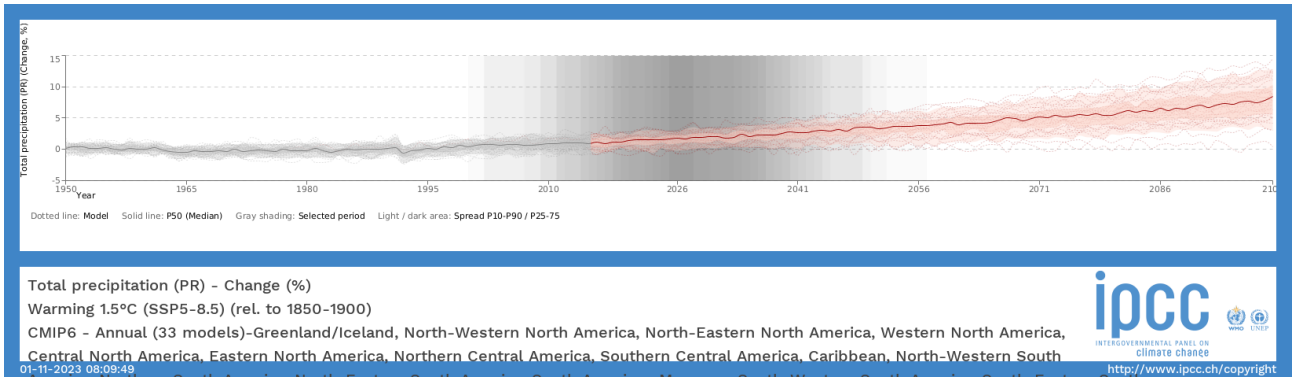


Figure 36: Total precipitation of the World

The total precipitation displays different trends between the world and South East Asia. While, in South East Asia, the average amount of the change in rain fall amount always fluctuate around 0, in the whole world, that amount has leveled off in the past, but expected to increase faster to above 5% in the 2100. The reason for this increasing in the rate of the rainfall amount might be the fast increasing in temperature at the end of the century.

13 Conclusion

Throughout this course, we've delved into a powerful numerical technique known as Finite Difference Methods (FDM). It's like a clever math trick, using Taylor theorem, to estimate values using polynomials. With FDM, we tackled partial differential equations (PDEs) related to things like advection and diffusion. We experimented with various schemes, such as forward, backward, centered, 4-order accurate, and the Runge-Kutta methods. These schemes help us simulate real-world situations. We explored their strengths and weaknesses, focusing on things like numerical modes and damping effects. We also got acquainted with the CFL criterion, a handy rule for ensuring our simulations behave well. And, of course, we had some fun exploring chaos theory, playing around with shallow water models, and peeking into global climate models.

I realize that there are still a lot of things that I can do like working more on optimizing the code, testing out more schemes, thinking more on the obtained results, but I did not spend enough time for it. However, after all, I am grateful that I have learned more about the mathematics, physics and modeling, especially the finite difference methods.

Finally, I want to say thank you to professor Ngo Duc Thanh for all the lectures/practices provided in this course. Even though I am not planning to work on climate, I am sure that the maths, physics and the modeling skill will be helpful for my future works in astrophysics.