Mech 587 – Computational Fluid Dynamics

Problem Set 1

Presented to Dr. Rajeev K. Jaiman

By Christian Rowsell (40131393)

2022-09-26

Contents

[Table of Figures 2](#_Toc115123073)

[Question 1 – Shallow Water Waves (15 Marks) 3](#_Toc115123074)

[Question 2 – 2D Steady, Inviscid, Incompressible Flow (20 Marks) 3](#_Toc115123075)

[Question 3 – Assessing Accuracy of ODE Integration (15 Marks) 4](#_Toc115123076)

[Appendix: 10](#_Toc115123077)

[Question 3 Code 10](#_Toc115123078)

# Table of Figures

[Figure 1: Part 1, dt=0.1 5](#_Toc115123056)

[Figure 2: Part 1, dt=0.2 6](#_Toc115123057)

[Figure 3: Part 1, dt=0.4 7](#_Toc115123058)

[Figure 4: Part 1, dt=0.8 8](#_Toc115123059)

[Figure 5: Part 2, Absolute Error Comparison 9](#_Toc115123060)

# Question 1 – Shallow Water Waves (15 Marks)

The shallow-water equations describe a thin layer of fluid of constant density in hydrostatic

balance, bounded from below by the bottom topography and from above by a free

surface.

To explore the mathematical structure of the shallow-water equations, consider the

following one-dimensional form of the time-dependent shallow water equations (Saint-

Venant equations):

where h denotes the spatial distribution of height of free water surface in a stream with

the velocity component u, and g represents the force acting on the fluid due to gravity.

Express the above system in a matrix form, find the eigenvalues, and show that the

system is hyperbolic.

# Question 2 – 2D Steady, Inviscid, Incompressible Flow (20 Marks)

The equations governing the steady, two-dimensional motion of an inviscid, incompressible

fluid (ρ = const) are:

Show that these equations always have just one real eigenvalue, and hence one characteristic

equation. Find the characteristic equation.

HINT: Start by transforming the equations into the matrix form ux + Auy = 0, where

u = (u, v, p)T .

# Question 3 – Assessing Accuracy of ODE Integration (15 Marks)

Consider the following initial value problem of first-order ODE system:

Forward Euler can be represented as the following,

where gn represents the function at the current timestep. Inputting the given function into the previous equation, and rearranging results in the following discretization using the Forward Euler method.

Backward Euler can be represented as the following,

where gn+1 represents the function at the next timestep. Inputting the given function into the previous equation, and rearranging to isolate un+1 results in the following discretization using the Backward Euler Method.

Finally, Trapezoidal rule can be represented as the following.

Where gn represents the function at the current timestep, and gn+1 represents the function at the next timestep. Inputting the given function into the previous equation, and rearranging to isolate un+1 results in the following discretization using the Trapezoidal rule.

Finally, the given ODE is an ODE with a known analytical solution. In this case, the analytical solution is given by

A Python script was written which compares all these different discretization methods, and answers the questions given below. The full code will be provided within the appendix.

1. (Stability) Plot the ODE solutions until final time tfinal = 8 obtained using the

forward Euler, the backward Euler and the Trapezoidal time integration schemes

at four representative values of time step sizes Δt = {0.1, 0.2, 0.4, 0.8}, and compare

the solutions obtained with the exact solution on the same plot. Briefly comment

on the results obtained.

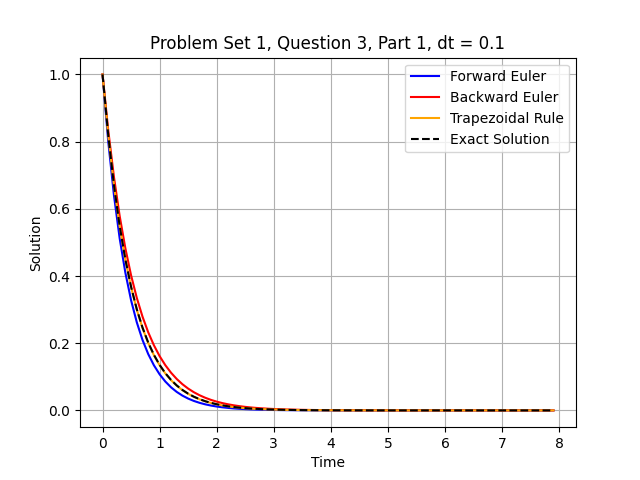


Figure 1: Part 1, dt=0.1

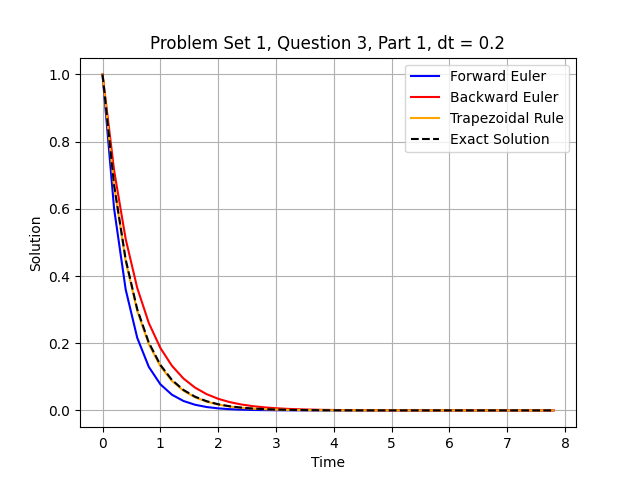


Figure 2: Part 1, dt=0.2

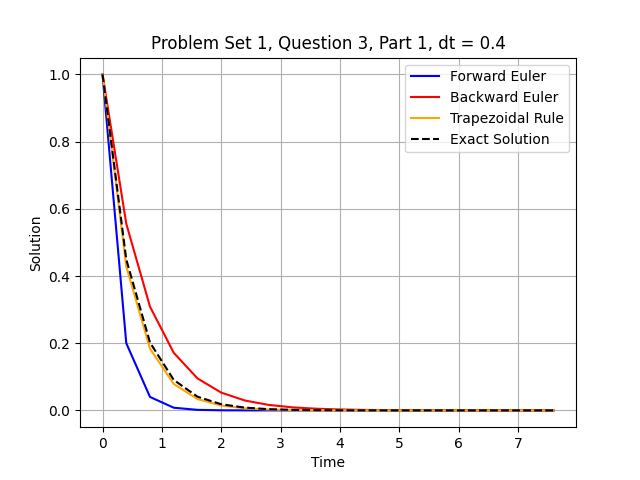


Figure 3: Part 1, dt=0.4

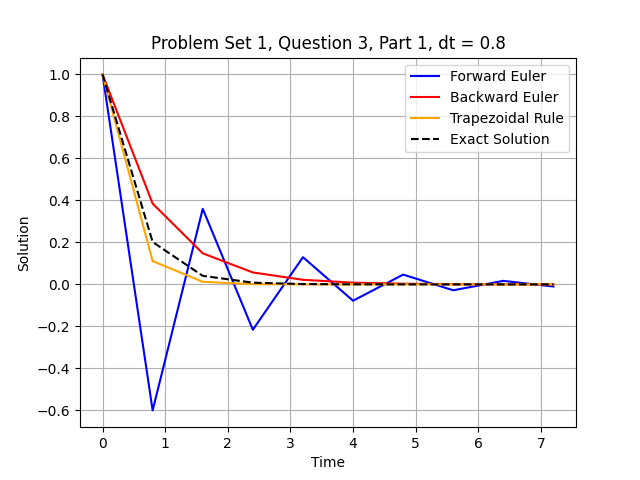


Figure 4: Part 1, dt=0.8

Figure 1, through Figure 4 show how each discretization method behaves as the timestep is changed, ranging from 0.1 to 0.8. Looking at Figure 1, the behaviour with a timestep of 0.1 can be found. At this timestep, all 3 methods represent the exact solution very well, with the Trapezoidal rule being the best, while the Forward Euler and Backward Euler methods respectively undershoot, and overshoot the exact solution. Figure 2 and Figure 3 represent timesteps of 0.2 and 0.4 respectively. It can be seen that the solutions begin to diverge as the timestep is increased with all 3 methods, with Forward Euler again undershooting, and Backward Euler again overshooting the exact solution. The Trapezoidal rule remains the most accurate of the 3. Finally, Figure 4 demonstrates the behaviour of the 3 methods with a timestep of 0.8. The Backward Euler method again overshoots the exact solution, and both the Backward Euler and Trapezoidal rule show a lower accuracy than at a timestep of 0.4, however they both still follow the general trend. The Forward Euler method however, shows a very different behaviour. With such a large timestep, the Forward Euler method becomes unstable, oscillating around the exact solution. This demonstrates the risk taken with the Forward Euler method, as if the timestep is not selected properly, this discretization method will become unstable.

1. (Order of accuracy) By varying step sizes (Δt), one can obtain different values of the

absolute local error at a particular time instant. Vary the step size Δt ∈ [0.001, 1],

and graphically show the absolute local error at t = 4.0 for the backward Euler and

Trapezoidal method. Briefly comment on the results obtained.

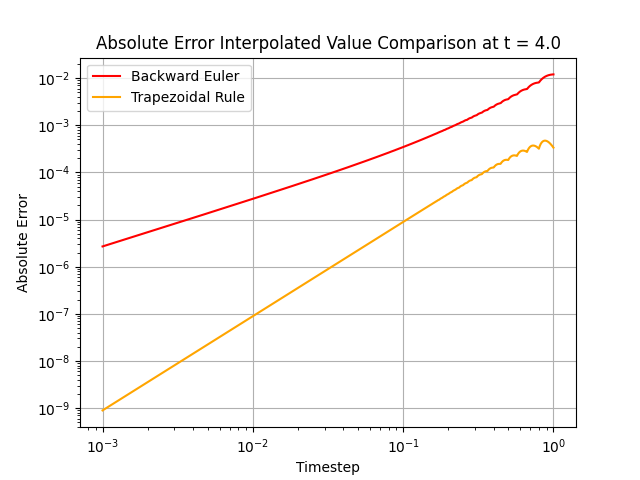


Figure 5: Part 2, Absolute Error Comparison

In order to maximize the number of data points taken for this question, an interesting approach was taken. If 4.0 was not a multiple of the timestep selected, the time marching loop would never reach a value where t was 4 exactly. It would always come close, but never reach 4 exactly. As such, there were 2 possible solutions. The first was only select timestep values that had 4 as a multiple, or the second was keep values that only came close, and use the neighboring values to interpolate the error at t=4. In order to maximize data points, the second method was selected. To see how exactly this was done, refer to the code in the appendix.

Referring to Figure 5, the absolute error curves can be seen for both the Backwards Euler and the Trapezoidal rule, plotted in a log-log scale. It can be seen that the slope of the Trapezoidal rule is greater than that of the Backwards Euler method, at about 2 for the Trapezoidal rule vs about 1 for the Backwards Euler method. This indicates that the Backwards Euler method has a time accuracy of O(dt), while the Trapezoidal rule has a time accuracy of O(dt2). Knowing this, it is recommended to select the Trapezoidal rule over the Backwards Euler, as a similar accuracy can be obtained with a larger timestep. This means that either the accuracy may be improved without sacrificing on computational cost, or the computational cost can be made lower while keeping the same accuracy.

For the order of accuracy analysis, you need to use log-scale for both the step size (along

horizontal X-axis) and the absolute error (along vertical Y-axis).

Note:

• Please append a screenshot of your code for problem 3 in your solution.

• All assignments should be submitted through Canvas.

# Appendix:

## Question 3 Code

# Import Packages  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy.interpolate import interp1d  
import os  
# Import User Defined Functions  
  
# Mech 587 - CFD  
# Christian Rowsell (40131393)  
  
# Find path to save files  
current\_path = os.getcwd()  
plot\_folder = current\_path + '/' + 'Part\_A'  
  
start\_t = 0 # first t value  
final\_t = 8 # Last t value used  
  
  
# Define function  
def q3(dt, tinitial, tfinal):  
 # Initialize arrays  
 nstep = (tfinal - tinitial) / dt  
 u1 = np.zeros(int(nstep))  
 u2 = np.zeros(int(nstep))  
 u3 = np.zeros(int(nstep))  
 e1 = np.zeros(int(nstep))  
 e2 = np.zeros(int(nstep))  
 e3 = np.zeros(int(nstep))  
 uex = np.zeros(int(nstep))  
 t = np.zeros(int(nstep))  
  
 # Initial conditions  
 t[0] = tinitial  
 u1[0] = 1  
 u2[0] = 1  
 u3[0] = 1  
 uex[0] = 1  
 # Solve equations  
 for i in range(1, int(nstep)):  
 u1[i] = u1[i-1] - 2\*dt\*u1[i-1] # Forward Euler  
 u2[i] = u2[i-1] / (1.0 + 2\*dt) # Backward Euler  
 u3[i] = u3[i-1] \* (1.0 - dt) / (1.0 + dt) # Trapezoidal Rule  
 t[i] = t[i-1] + dt  
 uex[i] = np.exp(-2\*t[i])  
  
 # If code below works  
 # if i == 4:  
 # err1 = abs(uex[i] - u1[i]) # Forward Euler  
 # err2 = abs(uex[i] - u2[i]) # Backward Euler  
 # err3 = abs(uex[i] - u3[i]) # Trapezoidal Rule  
 # return [err1, err2, err3]  
  
 # # Error Analysis  
 e1[i] = abs(uex[i] - u1[i]) # Forward Euler  
 e2[i] = abs(uex[i] - u2[i]) # Backward Euler  
 e3[i] = abs(uex[i] - u3[i]) # Trapezoidal Rule  
 #  
 # # Interpolate equations to allow finding of error analysis at t=4 exactly  
 eqn1 = interp1d(t, e1)  
 eqn2 = interp1d(t, e2)  
 eqn3 = interp1d(t, e3)  
 # Finding Absolute Error  
 err1 = eqn1(4)  
 err2 = eqn2(4)  
 err3 = eqn3(4)  
  
  
  
 # Plot Equations  
 # plt.plot(t, u1, label='Forward Euler', color='blue')  
 # plt.plot(t, u2, label='Backward Euler', color='red')  
 # plt.plot(t, u3, label='Trapezoidal Rule', color='orange')  
 # plt.plot(t, uex, label='Exact Solution', linestyle='--', color='black')  
 # plt.grid()  
 # plt.legend(loc='best')  
 # plt.xlabel('Time')  
 # plt.ylabel('Solution')  
 # title = 'Problem Set 1, Question 3, Part 1, dt = ' + str(dt)  
 # plt.title(title)  
 # # Save Files  
 # if not os.path.exists(plot\_folder):  
 # os.makedirs(plot\_folder, exist\_ok=True)  
 # plt.savefig(plot\_folder + '/' + title + '.png', bbox\_extra\_artists='legend\_outside')  
 # plt.show()  
 # plt.close()  
 # Dont return if code below works  
 return err1, err2, err3  
  
# Part 1  
# dt\_list = [0.1, 0.2, 0.4, 0.8]  
# for dt in dt\_list:  
# q3(dt, 0, 8)  
  
# Part 2  
# Code to find values for dt that will always pass 4  
  
'''  
DIDNT WORK  
# value = [] # Placedholder  
# dt\_list = [] # Values that will pass 4  
# increment = 0.0001  
# end = finalt / increment  
# for i in range(1, int(end)):  
# value.append(i \* increment) # Create list of all possible time steps based on increment  
# for j in range(len(value)):  
# if (4 / value[j]) % 1 == 0:  
# dt\_list.append(value[j])  
# If 4 divided by given time step is integer, 4 will be passed  
# and as such is stored in the array of suitable time steps  
# err = []  
for i in range(len(dt\_list)):  
 err.append(q3(dt\_list[i], start\_t, final\_t))  
'''  
  
dt\_list = np.linspace(0.001, 1, 10000) # Take 10000 equally spaced divisions between 0 and 1  
err1 = np.zeros(len(dt\_list))  
err2 = np.zeros(len(dt\_list))  
err3 = np.zeros(len(dt\_list))  
  
for i in range(len(dt\_list)):  
 err1[i], err2[i], err3[i] = q3(dt\_list[i], start\_t, final\_t)

# plt.plot(dt\_list, err1, label='Forward Euler', color='blue')  
plt.plot(dt\_list, err2, label='Backward Euler', color='red')  
plt.plot(dt\_list, err3, label='Trapezoidal Rule', color='orange')  
plt.grid()  
plt.title('Absolute Error Interpolated Value Comparison at t = 4.0')  
plt.legend(loc='best')  
plt.xlabel('Timestep')  
plt.ylabel('Absolute Error')  
plt.yscale('log')  
plt.xscale('log')  
plt.show()