MS6021 Scientific Computation

Assignment 5

Dara Corr - 22275193

November 26, 2022

Contents

1 Introduction

In this assignment we will be revising Quadrature and Explicit Methods of solving ODEs that we did in previous assignments but implementing the MATLAB codes we had into Python code.

The last part of this assignment looks at solving the Heat Equation numerically and determining for what time steps values give Stable or Unstable solutions to the Heat Equation.

2 Revision of Quadrature

We consider the following Integral:

$$I(\alpha) = \int_0^1 x^\alpha (2 + 3x^3) dx, \text{ for } \alpha > -1 \tag{1}$$

First, we integrate and evaluate this integral analytically.
Analytical solution:

$$\begin{aligned} I(\alpha) &= \int_0^1 x^\alpha (2 + 3x^3) dx, \text{ for } \alpha > -1 \\ &= \int_0^1 2x^\alpha + 3x^{\alpha+3} dx \\ &= \left[\frac{2x^{\alpha+1}}{\alpha+1} + \frac{3x^{\alpha+4}}{\alpha+4} \right] \Bigg|_{x=0}^{x=1} \\ &= \frac{2}{\alpha+1} + \frac{3}{\alpha+4} \end{aligned}$$

I rewrote my `trapezoidal` function from MATLAB into Python. This function calculates an area under a curve f(x) from lower bound a to upper bound b, using the Trapezoidal rule.

I had to make minor changes to my MATLAB code so that I could implement this into Python. The main changes I made were using `np.arange()` to create the array of values and I also made slight alterations to how the arrays were indexed in the code since `Python` indexing begins at 0 and not 1 like in MATLAB.

As we saw in the previous assignment, the Trapezoidal Rule is a method for approximating a definite integral by splitting the area under a curve $f(x)$ into Trapezoids of width $h = \frac{b-a}{N}$. The total Area under the curve is then calculated by summing the Areas of all the trapezoids.
Area of one trapezoid:

$$\begin{aligned} \text{Trapezoid Area } &= \text{Area of Square } + \text{Area of Triangle} \\ &= (h \cdot y_i) + \frac{1}{2} h(y_{i-1} - y_i) \\ &= \frac{h}{2}(y_i + y_{i-1}) \end{aligned}$$

Total Area under the curve:

$$\begin{aligned} \text{Area } &= \sum_{i=1}^{n} \frac{h}{2}(y_i + y_{i-1}) \\ &= \frac{h}{2}(y_0 + y_1) + \frac{h}{2}(y_1 + y_2) + ... + \frac{h}{2}(y_{n-1} + y_n) \\ &= \frac{h}{2}[y_0 + 2(y_1 + y_2 + ... + y_{n-1}) + y_n] \end{aligned}$$

Here is the code I used to implement the Trapezoidal Rule into `Python`:

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

def trapezoidal(f,a,b,N):
    h = (b-a)/N

    values = np.arange(a,b,h)
    np.delete(values, 0)
    np.delete(values, -1)

    middle = 2* sum(f(values))
    result = (h/2) * (f(a) + middle + f(b));
    return result
```

Then I considered a more general form of the integral

$$I_g = \int_0^1 x^\alpha g(x)dx \text{ for } \alpha > -1 \tag{2}$$

To solve this integral numerically, I created a new Python function called `tailored` which evaluates this integral numerically using a tailored version of the Trapezoidal Rule which uses the linear interpolant of $g(x)$ to compute the area under a curve traced out by $x^\alpha g(x)$ The Tailored Integral has the following form:

$$I_g \approx \sum_i \int_{x_{i-1}}^{x_i} x^\alpha g^I(x)dx \tag{3}$$

here $g^I(x)$ is the Linear Interpolant of $g(x)$ on the interval (x_{i-1}, x_i)
We can define the linear interpolant on the interval (x_{i-1}, x_i) as:

$$\frac{y - y_{i-1}}{x - x_{i-1}} = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

and if we solve for y:

$$y = y_{i-1} + (x - x_{i-1})\frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

$$\vdots$$

writing y as g(x): $y^I = g^I(x) = \dfrac{g(x_{i-1})(x_i - x) + g(x_i)(x - x_{i-1})}{x_i - x_{i-1}}$

Then,

$$I_g \approx \sum_i \int_{x_{i-1}}^{x_i} x^\alpha g^I(x)dx = \sum_i \int_{x_{i-1}}^{x_i} x^\alpha \frac{g(x_{i-1})(x_i - x) + g(x_i)(x - x_{i-1})}{x_i - x_{i-1}} dx$$

$$= \sum_i \int_{x_{i-1}}^{x_i} x^\alpha \frac{g(x_{i-1})(x_i) - x \cdot g(x_{i-1}) + x \cdot g(x_i) - x_{i-1} g(x_i)}{x_i - x_{i-1}} dx$$

$$= \sum_i \int_{x_{i-1}}^{x_i} x^\alpha \left[\frac{(x_i)g(x_{i-1}) - (x_{i-1})g(x_i)}{x_i - x_{i-1}} + \frac{g(x_i)x - g(x_{i-1})x}{x_i - x_{i-1}} \right] dx$$

$$= \sum_i \left[\int_{x_{i-1}}^{x_i} x^{\alpha+1} \frac{g(x_i) - g(x_{i-1})}{x_i - x_{i-1}} dx + \int_{x_{i-1}}^{x_i} x^\alpha \frac{(x_i)g(x_{i-1}) - (x_{i-1})g(x_i)}{x_i - x_{i-1}} dx \right]$$

$$= \sum_i \left[\frac{({x_i}^{\alpha+2} - {x_{i-1}}^{\alpha+2})}{\alpha + 2} \left(\frac{g(x_i) - g(x_{i-1})}{x_i - x_{i-1}} \right) + \frac{({x_i}^{\alpha+1} - {x_{i-1}}^{\alpha+1})}{\alpha + 1} \left(\frac{x_i \cdot g(x_{i-1}) - x_{i-1} \cdot g(x_i)}{x_i - x_{i-1}} \right) \right]$$

I implemented the above result into the function *tailored* in Python as follows:

```
def tailored(g,alpha,N):
    start = 0
    finish = 1
    h = (finish-start)/N

    x_i = np.linspace(start = (start+h),stop = finish, num = N)
    x_i_minus_one = np.linspace(start = start,stop = (finish-h)
        ,num = N)

    vals = lambda x: (g(x_i) - g(x_i_minus_one))/(x_i -
        x_i_minus_one) * (1/(alpha + 2)) * (x**(alpha + 2)) + (
        x_i*g(x_i_minus_one) -(x_i_minus_one)*(g(x_i)) )/(x_i -
        x_i_minus_one) * (1/(alpha + 1)) * (x**(alpha + 1))

    result = sum(vals(x_i) - vals(x_i_minus_one))
    return result
```

Then I computed Integral (1) for α = 1.2, 0.3 and -0.9 using trapezoidal and tailored functions and created a table with the error values $|I - I^{num}|$ between the Numerical Methods and the Analytical result.

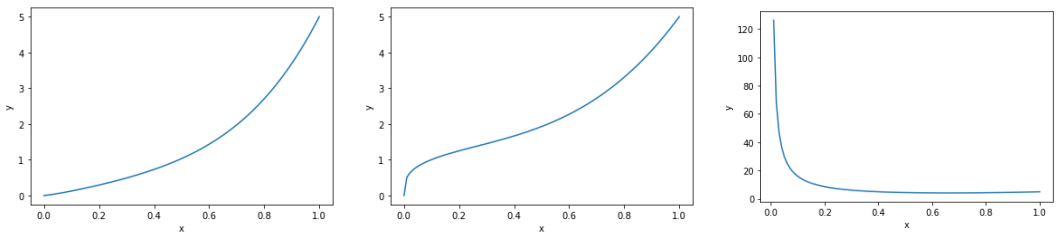
Table of Errors $|I - I^{num}|$:

	$\alpha = 1.2$	$\alpha = 0.3$	α = -0.9
Trapezoidal (N=10)	1.1796e-02	2.0704e-02	INF
Trapezoidal (N=100)	1.2064e-04	1.3886e-03	INF
Trapezoidal (N=10000)	1.2326e-08	3.6989e-06	INF
Tailored (N=10)	4.6770e-03	6.5154e-03	1.3986e-02
Tailored (N=100)	4.6874e-05	6.5217e-05	1.3664e-04
Tailored (N=10000)	4.6874e-09	6.5215e-09	1.3636e-08

(4)

2.1 Discussion and Conclusions

I plotted the three plots for $f(x) = x^{\alpha}(2 + 3x^3), \alpha = (1.2, 0.3, -0.9)$ to gain insight into how the two numeric integration methods performed.



$\alpha = 1.2$ $\alpha = 0.3$ $\alpha = -0.9$

Figure 1: Plots of $f(x) = x^{\alpha}(2 + 3x^3)$ against x for $\alpha = 1.2, \alpha = 0.3, \alpha = -0.9$

From the table (4), we see that the Tailored function outperforms the Trapezoidal function in all three cases.

The errors are closest between Trapezoidal and Tailored for the case $\alpha = 1.2$ where there is no asymptotes and no large changes in slope at any point in the graph.

For $\alpha = 0.3$, we notice a large change in slope just after the point $x = 0$. We also note that the Tailored function performs about the same as in the previous example but the performance of the Trapezoidal function has dropped for larger values of N.

For $\alpha = -0.9$, we notice that there is a vertical asymptote of $y \to +\infty$ near the point $x = 0$. The Trapezoidal function gives INF values for the error as it evaulates f(0) = +INF and this means the result here using Trapezoidal is an INF value. This is because of a vertical asymptote of $+\infty$ at $x = 0$, as we can see from the plot.

The Tailored function is able to overcome the problem with the asymptote and give reasonably small error values since it avoids calculating $y_0 = y(x_0) = (\frac{2}{x^{0.9}} + 3x^{2.1})\big|_{x=0}$ which

gives $y = +\infty$ at $x = 0$ using Trapezoidal rule. Instead, using the Linear Interpolant, it evaulates the slope at $x = 0$, which avoids returning Infinity values.

In conclusion, This Tailored method I created generates very precise results for our problem and outperformed the trapezoidal rule in every case we considered. Its precision is due to the fact that it uses the linear interpolant in calculating the area under the curve, which is more accurate than splitting the curve into trapezoids. The Trapezoidal rule does not perform as well as the Tailored method but manages to provide reasonable estimates of the integral results, especially for larger N. The trapezoidal method tends to underestimate the area under the curve when the curve is concave down and it tends to overestimate the area when the curve is concave up. The Trapezoidal rule gives exact results for linear functions and gives better results the more linear the plot is. The Trapezoidal rule fails for smooth functions which have points with large changes in slope Δy for a small change in x Δx, whereas Tailored (and quadratic integration methods like Simpson's rule) are able to evalutate slopes that do not behave approximately linearly within an interval h.

---

3 Revision of Explicit Methods of solving ODEs

In this part of the assignment, we will be implementing into `Python`, the Explicit Euler and Predictor Corrector methods that we used in Assignment 3.

We will be using them to solve problem 1 from assignment 3 and we will evaluate the performance of each method by finding the maximum errors between the computed solutions and the analytical solution for different values of N.

3.1 Outline of Explicit Methods Used:

We will be applying three Explicit Methods our problems in this assignment. The methods we will be using are the Explicit Euler Method and The Predictor Corrector Method. We will

be applying these methods to problem 1 from Assignment 3, given in the form $\frac{d}{dt}y = f(y,t)$ below:

$$\begin{aligned} \frac{d}{dt}y(t) &= -y(t) - 5e^{-t}sin(5t) \\ y(0) &= 1 \end{aligned} \tag{5}$$

This Equation has an analytical solution of $y(t) = e^{-t}cos(5t)$. We will use this analytical solution to compute errors from using the explicit ODE methods to solve the ODE.

The explicit methods we used have the following form:
Explicit Euler Method

$$\begin{gathered} y^{(0)} = y(0) \\ y^{(n+1)} = y^{(n)} + hf(t^{(n)}, y^{(n)}) \\ \text{where } t^{(n)} = nh, \text{ n} = \text{number of iterations} \end{gathered} \tag{6}$$

Predictor Corrector Method

$$\begin{gathered} y^{(0)} = y(0) \\ y^{*} = y^{(n)} + hf(t^{(n)}, y^{(n)}) \\ y^{(n+1)} = y^{(n)} + 0.5h[f(t^{(n)}, y^{(n)}) + f(t^{(n+1)}, y^{*})] \\ \text{where } t^{(n)} = nh \end{gathered} \tag{7}$$

I implemented these methods into Python as follows:

Explicit Euler Method:

```
#user provides function f, tspan, y0 and N

    h = (tspan[-1] - tspan[0])/N;

    y = np.zeros(N+1);

    #T = tspan(1):h:tspan(end);
    T = np.linspace(tspan[0],tspan[-1],N+1)

    y[0] = y0;

    for j in range(0,N):
        y[j+1] = y[j] + h* f(T[j],y[j]);


    Ysol = y
    Tsol = T
    return [Tsol, Ysol]
```

Predictor-Corrector Method:

```
#user provides function f, tspan, y0 and N

    h = (tspan[-1] - tspan[0])/N;

    ypred = np.zeros(N+1);
    y = np.zeros(N+1);

    #T = tspan(1):h:tspan(end);
    T = np.linspace(tspan[0],tspan[-1],N+1)

    y[0] = y0;
    ypred[0] = y0;

    for j in range(0,N):
        ypred[j+1] = y[j] + h* f(T[j],y[j]);
        y[j+1] = y[j] + (h/2) * ( f(T[j],y[j]) + f(T[j+1],ypred
            [j+1])  )


    Ysol = y
    Tsol = T
    return [Tsol, Ysol]
#%%
#Calculate Errors Explicit Euler
yprime = lambda t,y:  -y -5*np.exp(-t)*np.sin(5*t)
```

Then I created plots of the solutions and found the maximum error of both of the numerical solutions compared to the analytical solution.

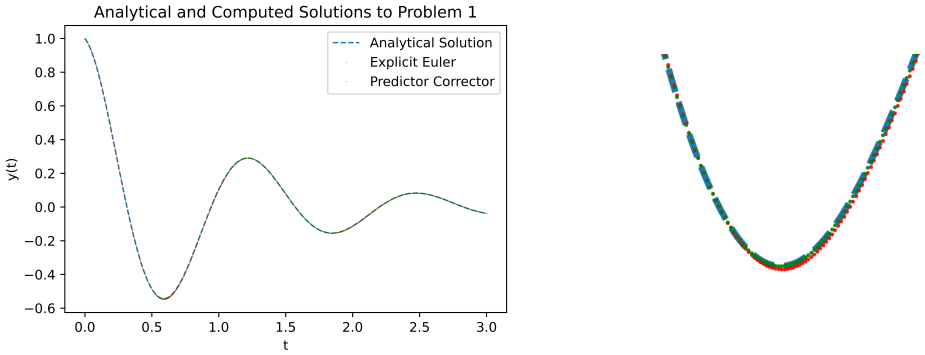


Figure 2: Plots of Numerical Solutions plotted against Analytical solution

Table of Errors:

	$N = 500$	$N = 1000$	$N = 2000$
Explicit-Euler	9.3567e-03	4.6796e-03	2.3402e-03
Predictor Corrector	9.3561e-05	2.3386e-05	5.8461e-06

3.2 Conclusions

We can see from the plots above and from the error table that the Predictor Corrector method is much more precise than the Explicit Euler Method. The explicit Euler method has relatively large errors to three decimal places while the Predictor-Corrector method gives reasonably precise errors in the magnitude of 5 to 6 decimal places using $N = 500, N = 1000$ and $N = 2000$ here. The Explicit Euler method performs quite poorly whereas the Predictor-Corrector method performs quite well and would be suitable for applications requiring results with good precision.

---

4 Parabolic PDEs: The Heat Equation

In This section we look to solve a parabolic partial differential equation numerically. We will be trying to solve the Heat Equation Numerically.

We will be considering the following well-posed PDE problem for the Heat Equation in x and t with $\alpha = 1$:

$$\begin{gathered} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \\ \text{for } 0 < x < 1,\, t > 0 \\ u(x,0) = sin(\pi x^2) \\ u(0,t) = u(1,t) = 0 \end{gathered} \tag{8}$$

To solve this problem numerically, it is first necessary to discretize it. I discretized it using a first order forward finite difference scheme in time for $\frac{\partial u}{\partial t}$ and then a second order centred finite difference scheme in x, for $\frac{\partial^2 u}{\partial x^2}$. I defined τ as the timestep and $h = 1/N$ as the step in the spatial variable x in this model. The resulting discretized equation is as follows:

$$\begin{gathered} \frac{U_n^{k+1} - U_n^k}{\tau} = \frac{U_{n+1}^k - 2U_n^k + U_{n-1}^k}{h^2} \\ \text{Where } n = 1, ..., N-1 \text{ and } k = 1, 2, ...K \end{gathered} \tag{9}$$

We can rearrange this to get a scheme in K for finding U_i^k for $k = 1, 2, ...K$:

$$U_n^{k+1} = \tau \left[\frac{U_{n+1}^k - 2U_n^k + U_{n-1}^k}{h^2} \right] + U_n^k \tag{10}$$

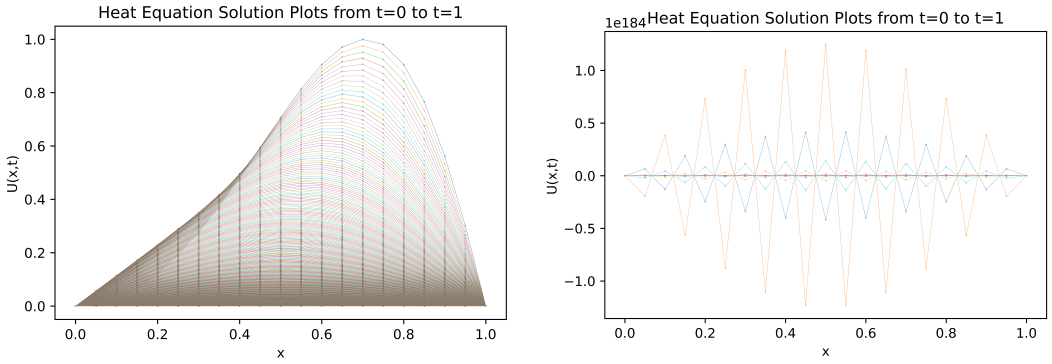
Using initial and boundary conditions defined in (8) to find $U_i^{k=0}$ and from this we can use our scheme (10) to find U_i^k for all x and t values.

I implemented this method into Python as follows for a number of different timesteps τ:

```
N = 40

h = 1/N

x = np.linspace(0,1, N+1)
print(x)

U_i =  np.sin(np.pi* x**2) #k = 0 here
U_i[-1] = 0; U_i[0] = 0; #Boundary Condition, U(x=N,k) = 0, U(x
    =0,k) = 0

#Define Tau
Tau = (h**2)/4

Kappa = round(1/Tau)

U_1 =  np.sin(np.pi* x**2) #U_{i=1}
U_1[-1] = 0; U_i[0] = 0;

plt.figure(dpi=1200)
plt.plot(x,U_1,'o-',linewidth = 0.2, markersize=0.2)

#we want to fill grid with U_i values for every time step k
for K in range(0,(Kappa+1)):
    U_i_k_plus_one = Tau * (U_i[2:(N+1)] -2*U_i[1:N] + U_i[0:(N
        -1)])/(h**2) + U_i[1:N]
    U_i_k_plus_one = np.append(0,U_i_k_plus_one)
    U_i_k_plus_one = np.append(U_i_k_plus_one,0)
    plt.plot(x,U_i_k_plus_one,'o-', linewidth = 0.2, markersize
        =0.2 ,alpha = 0.8,)
    U_i = U_i_k_plus_one


plt.title("Heat Equation Solution Plots from t=0 to t=1")
```

```
plt.xlabel("x")
plt.ylabel("U(x,t)")
plt.show()
```

Here are some sample plots I generated with N set as $N = 20$ for $\tau = \frac{h^2}{2}$ and $\tau = h^2$:



$\tau = \frac{h^2}{2}$ $\tau = h^2$

Figure 3: Plots of Solutions to our Heat Equation Problem with stable solutions obtained for $\tau = \frac{h^2}{2}$ and unstable solutions obtained for $\tau = h^2$

We notice in the above plots that we obtain stable solutions for $\tau = \frac{h^2}{2}$ and we obtain unstable solutions for $\tau = h^2$.

I investigated this further for different N and τ values:

	$\tau = \frac{h^2}{4}$	$\tau = \frac{h^2}{2}$	$\tau = h^2$	$\tau = 2h^2$	$\tau = 4h^2$
N= 40	Stable	Stable	Unstable	Unstable	Unstable
N= 80	Stable	Stable	Unstable	Unstable	Unstable
N= 200	Stable	Stable	Unstable	Unstable	Unstable

4.1 Conclusions

Using this method to numerically solve the Heat Equation (8), I found that the solutions are stable so long as $\frac{\tau}{h^2} \leq \frac{1}{2}$.

This result can be Derived using Von Neumann stability analysis and the stability condition for this problem can be written more generally as

$$r = \frac{\alpha \Delta t}{\Delta x^2} \leq \frac{1}{2}$$

From this result, I would choose τ such that $\tau \leq \frac{h^2}{2}$ here, so that we can obtain stable solutions.