

# Homework 3

Due Date: October 27, 2025

## 1 - Hermite Interpolation

Given the function

$$f(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1],$$

with the function values and first derivatives at the nodes

$$x_0 = -1, \quad x_1 = 0, \quad x_2 = 1$$

### Question 1.1

Construct the cubic Hermite interpolant  $H(x)$  and evaluate  $H(0.5)$ .

*Solution.* The cubic Hermite interpolant on an interval  $[x_0, x_1]$  is determined by the shape functions:

$$h_{00}(t) = 2t^3 - 3t^2 + 1, \quad h_{01}(t) = -2t^3 + 3t^2$$

$$h_{10}(t) = t^3 - 2t^2 + t, \quad h_{11}(t) = t^3 - t^2$$

Provided  $h = x_1 - x_0$  and  $t = \frac{x - x_0}{h} \in [0, 1]$ . The cubic Hermite interpolant is given by:

$$H(x) = f(x_0)h_{00}(t) + f(x_1)h_{01}(t) + h[f'(x_0)h_{10}(t) + f'(x_1)h_{11}(t)]$$

We will construct the Hermite interpolant piecewise on the intervals  $[-1, 0]$  and  $[0, 1]$ .

To start off with, the following table summarizes the function and derivative values at the nodes:

Node $x_j$	$f(x_j)$	$f'(x_j)$
-1	$\frac{1}{26}$	$\frac{50}{676} = \frac{25}{338}$
0	1	0
1	$\frac{1}{26}$	$-\frac{25}{338}$

Where  $f'(x) = \frac{-50x}{(1+25x^2)^2}$ .

On  $[-1, 0]$ , we have  $h = 1$  and  $t = x + 1$ . So therefore we have:

$$h_{00}(x + 1) = 2(x + 1)^3 - 3(x + 1)^2 + 1 = 2x^3 + 3x^2$$

$$h_{01}(x + 1) = -2(x + 1)^3 + 3(x + 1)^2 = -2x^3 - 3x^2 + 1$$

$$h_{10}(x + 1) = (x + 1)^3 - 2(x + 1)^2 + (x + 1) = x^3 + x^2$$

$$h_{11}(x+1) = (x+1)^3 - (x+1)^2 = x^3 + 2x^2 + x$$

Substituting into the interpolant formula, we get:

$$H(x) = \frac{1}{26}(2x^3 + 3x^2) + 1(-2x^3 - 3x^2 + 1) + 1 \cdot \frac{25}{338}(x^3 + x^2) + 0$$

Simplifying, we have:

$$H(x) = -\frac{625}{338}x^3 - \frac{475}{169}x^2 + 1, \quad x \in [-1, 0]$$

On  $[0, 1]$ , we have  $h = 1$  and  $t = x$ . So therefore our shape functions are as defined initially with  $x$  in place of  $t$ . Substituting into the interpolant formula, we get:

$$H(x) = 1(2x^3 - 3x^2 + 1) + \frac{1}{26}(-2x^3 + 3x^2) + 0 - \frac{25}{338}(x^3 - x^2)$$

Simplifying, we have:

$$H(x) = \frac{625}{338}x^3 - \frac{475}{169}x^2 + 1, \quad x \in [0, 1]$$

So the cubic Hermite interpolant is:

$$H(x) = \begin{cases} -\frac{625}{338}x^3 - \frac{475}{169}x^2 + 1, & x \in [-1, 0] \\ \frac{625}{338}x^3 - \frac{475}{169}x^2 + 1, & x \in [0, 1] \end{cases}$$

Evaluating at  $x = 0.5$ , we get that:

$$H(0.5) = \frac{625}{338}(0.5)^3 - \frac{475}{169}(0.5)^2 + 1 = \frac{1429}{2704} \approx 0.528479290$$

## Question 1.2

Evaluate the absolute error of  $f(0.5) - H(0.5)$ , and show that the error agrees the prior estimate of

$$f(x) - p(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{j=0}^n (x - x_j)^2$$

for some  $\xi$  between  $x$  and  $x_j$ .

*Solution.* Note that  $f(0.5) = \frac{4}{29} \approx 0.1379310345$ .

The absolute error with our estimate  $H(0.5)$  is:

$$|f(0.5) - H(0.5)| = \left| \frac{4}{29} - \frac{1429}{2704} \right| = \frac{30625}{78416} \approx 0.390548$$

The fourth derivative of  $f(x)$  is:

$$f^{(4)}(x) = \frac{15000(3125x^4 - 250x^2 + 1)}{(25x^2 + 1)^5}$$

The maximum of  $|f^{(4)}(\xi)|$  for  $\xi \in [-1, 1]$  occurs at  $x = 0$ , yielding:

$$|f^{(4)}(0)| = 15000$$

The product term evaluated at  $x = 0.5$  is:

$$\prod_{j=1}^2 (0.5 - x_j)^2 = (0.5 - 0)^2 (0.5 - 1)^2 = \left(\frac{1}{2}\right)^2 \cdot \left(-\frac{1}{2}\right)^2 = \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$$

We also have that  $(2n + 2)! = 4! = 24$  for  $n = 1$ . So putting everything together, we can estimate the error as:

$$|f(0.5) - H(0.5)| \leq \frac{15000}{24} \cdot \frac{1}{16} = \frac{625}{16} \approx 39.0625$$

So therefore our absolute error is well within the established bound.

#### Extra: Absolute Error with Standard Hermite Interpolant

*Note: Question 1.1 was originally done by hand with the regular Hermite Interpolant resulting in a polynomial of degree 5. This was the error analysis conducted for that version.*

The absolute error with our estimate  $P_5(0.5)$  is:

$$|f(0.5) - P_5(0.5)| = \left| \frac{4}{29} - \frac{6341}{10816} \right| = \frac{140625}{313664} \approx 0.4483300602$$

To estimate the error using the provided formula, we first need to compute the sixth derivative of  $f(x)$ , which works out to:

$$f^{(6)}(x) = \frac{11250000 (109375x^6 - 21875x^4 + 525x^2 - 1)}{(25x^2 + 1)^7}$$

The maximum of  $|f^{(6)}(\xi)|$  for  $\xi \in [-1, 1]$  occurs at  $x = 0$ , yielding:

$$|f^{(6)}(0)| = 11250000$$

The product term evaluated at  $x = 0.5$  is:

$$\prod_{j=0}^2 (0.5 - x_j)^2 = (0.5 + 1)^2 (0.5 - 0)^2 (0.5 - 1)^2 = \left(\frac{3}{2}\right)^2 \cdot \left(\frac{1}{2}\right)^2 \cdot \left(-\frac{1}{2}\right)^2 = \frac{9}{4} \cdot \frac{1}{4} \cdot \frac{1}{4} = \frac{9}{64}$$

We also have that  $(2n + 2)! = 6! = 720$  for  $n = 2$ .

So putting everything together, we can estimate the error as:

$$|f(0.5) - P_5(0.5)| \leq \frac{11250000}{720} \cdot \frac{9}{64} = \frac{140625}{64} \approx 2197.265625$$

So therefore our absolute error is well within the established bound.

## Question 1.3

Obtain the cubic Hermite interpolant in Newton form  $P(x)$ , by constructing the Hermite divided-difference table.

*Solution.* The code for Lagrange Interpolation from the previous assignment was modified compute the Hermite Interpolation:

Listing 1: 1.3 Python

```

1 import numpy as np
2
3 def f(x):
4     return 1 / (1 + 25 * x**2)
5
6 def f_prime(x):
7     return -50 * x / (1 + 25 * x**2)**2
8
9 def hermite_coefficients(nodes, f=f, f_prime=f_prime):
10     # Parameterize nodes for Hermite interpolation
11     z = np.concatenate((nodes, nodes))
12     sorted_indexes = z.argsort()
13     z = z[sorted_indexes]
14     num_nodes = len(z)
15
16     # Set up dd table with zeroth dd
17     dd_table = np.array([[f(zi) for zi in z]])
18
19     # First divided difference
20     f_prime_nodes = np.array([f_prime(xi) for xi in nodes])
21     zeros = np.zeros(len(nodes))
22     first_dd = np.concatenate((f_prime_nodes, zeros))[sorted_indexes]
23     for j in range(num_nodes - 1):
24         if j % 2 == 1:
25             first_dd[j] = (dd_table[0, j + 1] - dd_table[0, j]) / (z[j + 1] - z[j])
26     dd_table = np.vstack([dd_table, first_dd])
27
28     # Remaining Divided Differences
29     for i in range(2, num_nodes):
30         ith_dd = np.zeros(num_nodes)
31
32         for j in range(num_nodes - i):
33             # Calculate ith divided differences
34             ith_dd[j] = (dd_table[i - 1, j + 1] - dd_table[i - 1, j]) / (z[j + i] - z[j])
35
36         dd_table = np.vstack([dd_table, ith_dd])

```

```

37
38     coefficients = np.array([dd_table[i, 0] for i in range(dd_table.shape[0])])
39     return coefficients, dd_table.T, z
40
41 def generate_hermite(nodes, n):
42     # Get coefficients
43     a = hermite_coefficients(nodes)[0]
44     # Start with function and constant term
45     equation = f"P_{{{2*n+1}}}(x) = {a[0]}"
46     w = []
47     # Build (x - xi) terms
48     for xi in nodes:
49         if abs(xi) <= 1e-14:
50             w.append('(x)')
51         elif xi < 0:
52             w.append(f"(x + {abs(xi)})")
53         else:
54             w.append(f"(x - {xi})")
55     b = ""
56
57     # Build polynomial string
58     for i in range(1, len(a)):
59         for j in range(i):
60             # Multiply (x - xi) terms
61             b += '^2' if j%2 == 1 else w[int(j/2)]
62             # Multiply (x - xi) product with current coefficient and add term
63             if a[i] > 0:
64                 equation += f" + {a[i]}{b}"
65             elif a[i] < 0:
66                 equation += f" - {abs(a[i])}{b}"
67             b = ""
68
69     return equation
70
71 if __name__ == '__main__':
72     nodes_1 = np.array([-1,0])
73     equation_1 = generate_hermite(nodes_1, 1)
74     table_1, z_1 = hermite_coefficients(nodes_1)[1:3]
75
76     nodes_2 = np.array([0,1])
77     equation_2 = generate_hermite(nodes_1, 1)
78     table_2, z_2 = hermite_coefficients(nodes_1)[1:3]
79
80     equations = [equation_1, equation_2]
81     tables = [table_1, table_2]
82     z = [z_1, z_2]
83

```

```

84     with open("./outputs_3/hermite.txt", 'w') as file:
85         for k in range(2):
86             file.write(equations[k] + '\n\n')
87
88             file.write('\begin{center}\n')
89             file.write('\begin{tabular}{|c|c|c|c|c|}\n')
90             file.write('\hline\n')
91             file.write('& $z_i$ & $f[z_i]$ & 1st dd. & 2nd dd. & 3rd dd. \\\n
92
93         file.write('\hline\n')
94         for i in range(tables[k].shape[0]):
95             file.write(f'$z_{i}$ & ${z[k][i]}$ ')
96             for j in range(tables[k].shape[1]):
97                 file.write(f'& ${tables[k][i,j]:.4f}$ ')
98             file.write('\n')
99             file.write('\hline\n')
100            file.write('\end{tabular}\n')
101            file.write('\end{center}\n\n')

```

The divided-difference tables and Hermite Interpolants generated were:

On  $[-1, 0]$ :

	$z_i$	$f[z_i]$	1st dd.	2nd dd.	3rd dd.
$z_0$	-1	0.0385	0.0740	0.8876	-1.8491
$z_1$	-1	0.0385	0.9615	-0.9615	0.0000
$z_2$	0	1.0000	0.0000	0.0000	0.0000
$z_3$	0	1.0000	0.0000	0.0000	0.0000

$$\begin{aligned}
 P_3(x) = & 0.038461538461538464 + 0.07396449704142012(x + 1) \\
 & + 0.8875739644970414(x + 1)^2 - 1.849112426035503(x + 1)^2(x)
 \end{aligned}$$

On  $[0, 1]$ :

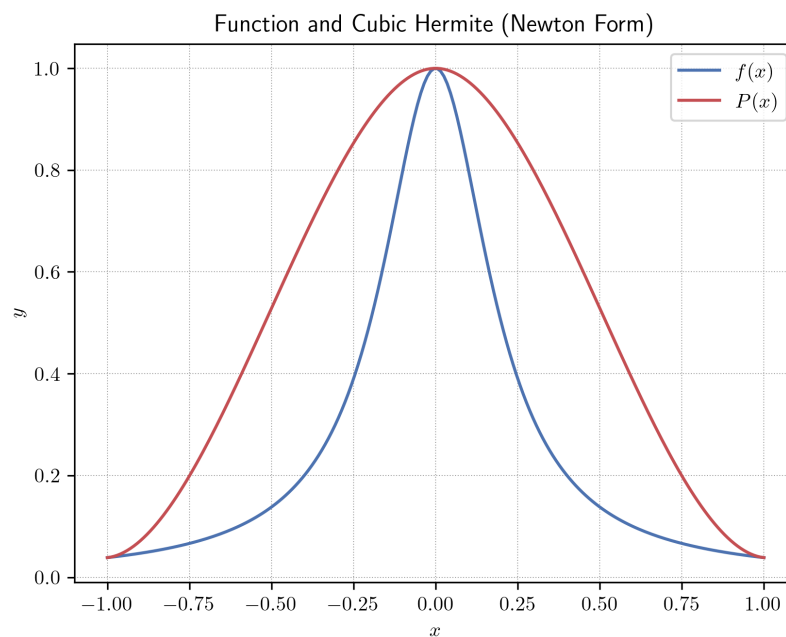
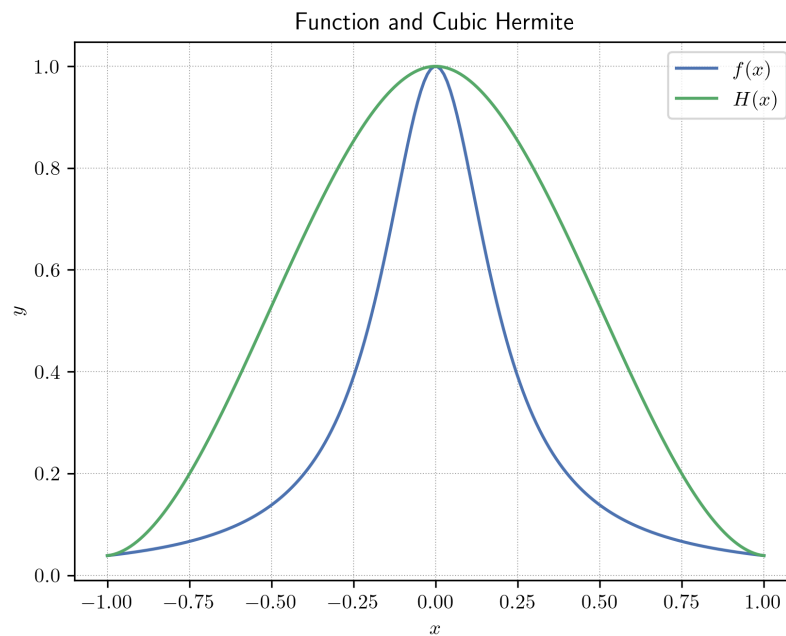
	$z_i$	$f[z_i]$	1st dd.	2nd dd.	3rd dd.
$z_0$	0	1.0000	0.0000	-0.9615	1.8491
$z_1$	0	1.0000	-0.9615	0.8876	0.0000
$z_2$	1	0.0385	-0.0740	0.0000	0.0000
$z_3$	1	0.0385	0.0000	0.0000	0.0000

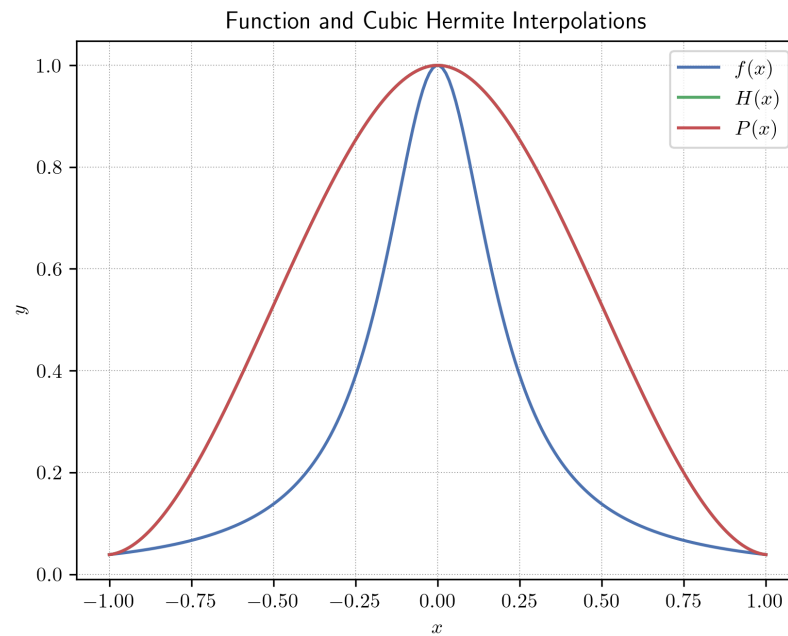
$$\begin{aligned}
 P_3(x) = & 1.0 - 0.9615384615384616(x)^2 \\
 & + 1.849112426035503(x)^2(x - 1)
 \end{aligned}$$

**Question 1.4**

Plot  $f(x)$ ,  $H(x)$  and  $P(x)$  on  $x \in [-1, 1]$ .

*Solution.* The following plots were generated:





We observe that both Hermite interpolants  $H(x)$  and  $P(x)$  are identical and loosely follow the behavior of  $f(x)$  on the interval  $[-1, 1]$ .



## 2 - Simple Analysis on Finite Difference

Consider the forward difference formula for the first derivative of a smooth function  $f(x)$ :

$$D_h f(x) = \frac{f(x+h) - f(x)}{h}$$

The total error for this approximation can be expressed as:

$$E(h) = \frac{C_1}{h} + C_2 h$$

with the optimal step size  $h$ :

$$h_{opt} = \sqrt{\frac{C_1}{C_2}}$$

where  $C_1$  and  $C_2$  are constants depending on  $f(x)$ , machine precision  $\varepsilon$ , and its derivatives.

Let  $f(x) = e^x$ , and take  $x = 1.5$ .

### Question 2.1

Compute  $f'(1.5)$  using the forward difference formula for a range of step sizes  $h = 10^{-k}$ , where  $k = 1, 2, \dots, 10$ .

*Solution.* The following code was used to compute the forward difference approximations:

Listing 2: 2.1 Python

```

1 import numpy as np
2
3 def forward_diff(f, x, step=1):
4     return (f(x + step) - f(x)) / step
5
6 if __name__ == "__main__":
7     with open("./outputs_3/forward_diff.txt", "w") as file:
8         for k in range(1,11):
9             h = 10**(-k)
10            f_prime = forward_diff(np.exp, 1.5, h)
11            file.write(f'\\[D_h^{k}\\]f(1.5) = {f_prime}\\[\\n')

```

The following results were obtained for  $D_h^{(k)} f(1.5)$ :

$$D_h^{(1)} f(1.5) = 4.713433540570504$$

$$D_h^{(2)} f(1.5) = 4.5041723976187775$$

$$D_h^{(3)} f(1.5) = 4.483930662008362$$

$$D_h^{(4)} f(1.5) = 4.481913162264206$$

$$D_h^{(5)} f(1.5) = 4.4817114789097445$$

$$D_h^{(6)} f(1.5) = 4.48169131139764$$

$$D_h^{(7)} f(1.5) = 4.481689304114411$$

$$D_h^{(8)} f(1.5) = 4.481689064306238$$

$$D_h^{(9)} f(1.5) = 4.481689686031132$$

$$D_h^{(10)} f(1.5) = 4.48169501510165$$

## Question 2.2

Plot the absolute error  $|D_h f(1) - f'(1)|$  versus  $h$  on a log-log scale. (Report code)

*Solution.* The following code was used to compute and plot the absolute error:

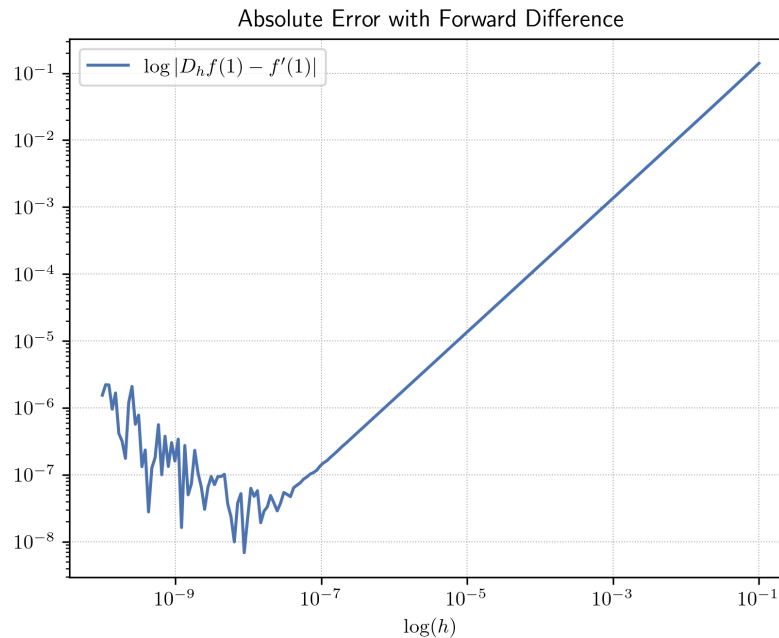
Listing 3: 2.2 Python

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def calc_absolute_error(f, step, eval_x=1, diff_method=forward_diff):
5     approx = diff_method(f, eval_x, step)
6     return np.abs(approx - f(eval_x))
7
8 if __name__ == "__main__":
9     # <Some matplotlib styling and enable LaTeX>
10    h = np.logspace(-1, -10, 200)
11    error = calc_absolute_error(np.exp, h)
12    plt.loglog(h, error, label='$\\log|D_{hf}(1) - f\\'(1)|$')
13    plt.xlabel('$\\log(h)$')
14    plt.title('Absolute Error in Forward Difference Approximation')
15    plt.legend()
16    plt.savefig('./outputs_3/forward_diff_graph.png', dpi=300)
17    plt.show()

```

The resulting plot was:



We observe that machine error starts to occur as  $h$  gets smaller than approximately than  $10^{-7}$ .

### Question 2.3

Identify the  $h$  that minimizes the total error and compare it with the predicted  $h_{opt}$ .

*Solution.* From the plot above, we observe that the minimum error occurs around  $h \approx 10^{-8}$ .

Machine error is  $\varepsilon \sim 10^{-16}$  for double precision floating point numbers. From the notes, we have that  $C_1 \sim \varepsilon |f(1)|$  and  $C_2 \sim \frac{|f''(1)|}{2}$ . This resulted in  $h_{opt} \propto \sqrt{\varepsilon} \approx 10^{-8}$ , which aligns with our observed minimum error point.

### Question 2.4

Use the one-sided three points scheme to approximate  $f'(1.5)$ , and repeat question 2.2.

*Solution.* The one-sided three-point finite difference formula for the first derivative is given by:

$$D_h f(x) = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}$$

The following code was used to compute and plot the absolute error using the three-point forward difference scheme:

Listing 4: 2.4 Python

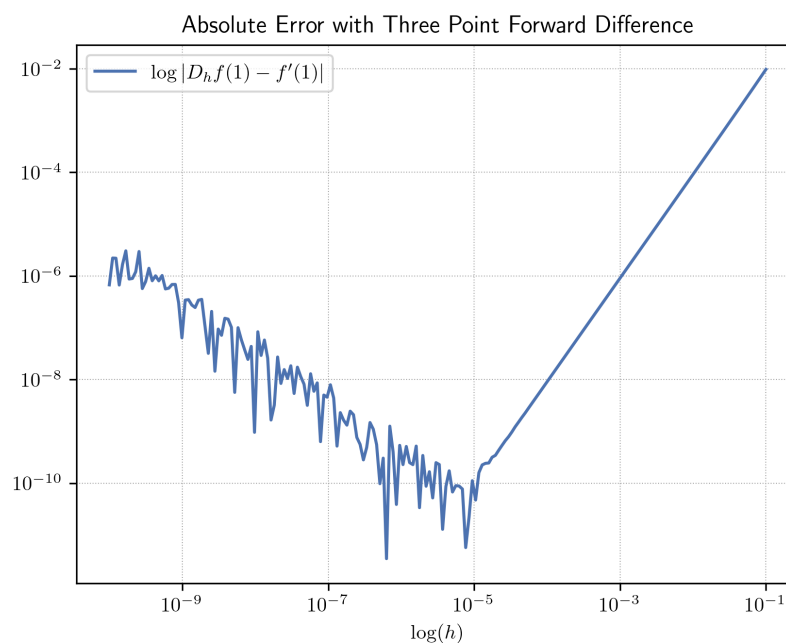
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def three_point_forward_diff(f, x, step=1):
```

```

5     return (-3 * f(x) + 4 * f(x + step) - f(x + 2 * step)) / (2 * step)
6
7 if __name__ == "__main__":
8     # <Some matplotlib styling and enable LaTeX>
9     h = np.logspace(-1, -10, 200)
10    error = calc_absolute_error(np.exp, h, diff_method=three_point_forward_diff
11    )
12
13    plt.loglog(h, error, label='$\\log|D_h f(1) - f'(1)|$')
14    plt.xlabel('$\\log(h)$')
15    plt.title('Absolute Error with Three Point Forward Difference')
16    plt.legend()
17    plt.savefig('./outputs_3/three_point_forward_diff_graph.png', dpi=300)
18    plt.show()

```

The resulting plot was:



We observe that machine error starts to occur as  $h$  gets smaller than approximately  $10^{-5}$ .

### 3 - Solve Burger's Equation by Finite Difference Scheme

Consider the 1D viscous Burgers' equation:

$$u_t + uu_x = \nu u_{xx}, \quad x \in [0, 1], \quad t > 0, \quad \nu = 0.2$$

Design, analyze, and implement a numerical method that is stable and achieves at least second-order accuracy in both space and time.

#### Question 3.1

Report finite difference scheme used.

*Solution.* We will be using RK4 with 2nd order finite differences for space.

First let us discretize our system. Let  $x_0, x_1, \dots, x_n$  be a partition of  $[0, 1]$  with step size  $\Delta x = h = \frac{1}{n}$ . Let  $t_0, t_1, \dots, t_m$  be a partition of  $[0, T]$  with step size  $\Delta t$ . Let  $u_i$  be the numerical approximation of  $u(x_i, t)$ .

We can construct our schemes for  $u_x, u_{xx}$  from the Taylor expansion of  $u$  centered at  $x_i$ :

$$u(x) = u(x_i) + u_x(x_i)(x - x_i) + \frac{u_{xx}(x_i)}{2}(x - x_i)^2 + \frac{u_{xxx}(x_i)}{6}(x - x_i)^3 + \dots$$

Which gives us for  $x = x_i - h$  and  $x = x_i + h$ :

$$u(x_i - h) = u(x_i) - hu_x(x_i) + \frac{h^2}{2}u_{xx}(x_i) - \frac{h^3}{6}u_{xxx}(x_i) + O(h^4)$$

$$u(x_i + h) = u(x_i) + hu_x(x_i) + \frac{h^2}{2}u_{xx}(x_i) + \frac{h^3}{6}u_{xxx}(x_i) + O(h^4)$$

Approximate  $u_x$  centrally by:

$$u(x_i + h) - u(x_i - h) = 2hu_x(x_i) + O(h^3)$$

Which gives us the second-order scheme:

$$u_x(x_i) \approx \frac{u(x_i + h) - u(x_i - h)}{2h} + O(h^2)$$

Approximate  $u_{xx}$  with the usual second-order central difference:

$$u(x_i + h) - 2u(x_i) + u(x_i - h) = h^2u_{xx}(x_i) + O(h^4)$$

Which gives us the second-order scheme:

$$u_{xx}(x_i) \approx \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2} + O(h^2)$$

Rearranging Burger's equation for  $u_t$ , we have:

$$u_t = -uu_x + vu_{xx}$$

Substituting in our finite difference approximations, we obtain the ODE:

$$\frac{du_i}{dt} \approx -u_i \cdot \frac{u_{i+1} - u_{i-1}}{2h} + v \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2)$$

Which is  $O(h^2) = O(\Delta x^2)$  accurate in space. Denote the right-hand side above as  $F(u_i)$ .

More compactly, write the system as:

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

Where  $u = [u_0, u_1, \dots, u_n]^T$ .

We will now use RK4 to approximate  $u^{n+1}$ , which is given by:

$$u^{n+1} = u^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Such that:

$$\begin{aligned} k_1 &= F(u^n) \\ k_2 &= F\left(u^n + \frac{\Delta t}{2}k_1\right) \\ k_3 &= F\left(u^n + \frac{\Delta t}{2}k_2\right) \\ k_4 &= F(u^n + \Delta tk_3) \end{aligned}$$

RK4 is  $O(\Delta t^4)$  accurate in time, so our overall scheme is  $O(\Delta t^4) + O(\Delta x^2)$ . We also note that RK4 would have CFL conditions for stability:

$$\Delta t \leq \frac{C_1 \Delta x}{\max |u|}, \quad \Delta t \leq \frac{C_2 \Delta x^2}{v}$$

Where  $C_1, C_2$  are constants depending on the scheme. In practice, we can choose  $\Delta t$  based on the more restrictive of the two conditions above and pick smaller  $C_1, C_2$  to be safe.

### Question 3.2

Use the initial condition as a sine wave  $\sin(2\pi x)$ . Implement the scheme and plot the numerical solution at  $T = 1$  second.

*Solution.* We have Burger's equation:

$$u_t + uu_x = vu_{xx}, \quad x \in [0, 1], \quad t > 0, \quad v = 0.2$$

With periodic initial condition:

$$u(x, 0) = \sin(2\pi x)$$

This means that  $u(0, t) = u(1, t)$  for all  $t \geq 0$ . Since our data is periodic, no extra scheme is needed for the boundaries and we can just treat  $u$  as a circular array.

The following code was used to implement the scheme:

Listing 5: 3.2 Python

```

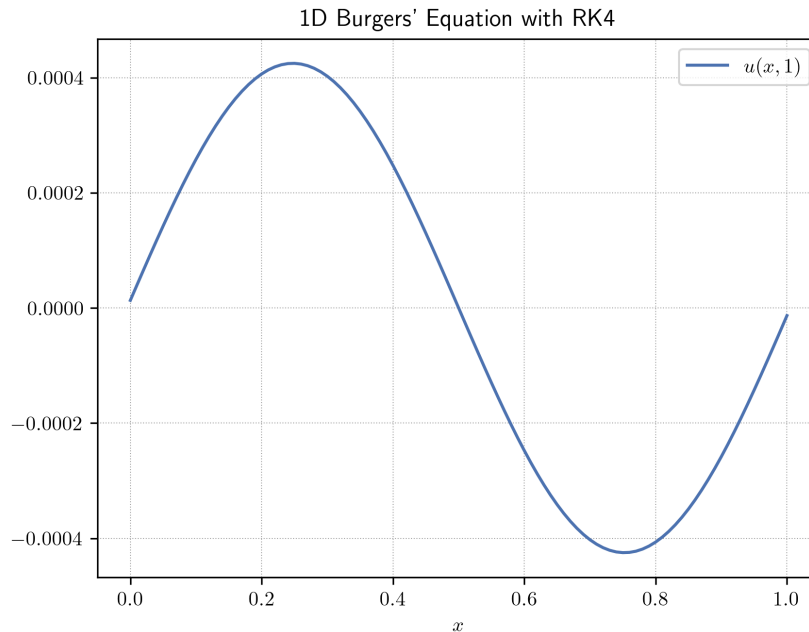
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from copy import deepcopy
4
5 def initial_condition(x):
6     return np.sin(2 * np.pi * x)
7
8 def rk4_step(u, dt, dx, v):
9     def F(u):
10         # Compute spatial derivatives using central differences
11         u_x = (np.roll(u, -1) - np.roll(u, 1)) / (2 * dx)
12         u_xx = (np.roll(u, -1) - 2 * u + np.roll(u, 1)) / (dx**2)
13         return -u * u_x + v * u_xx
14
15     k1 = F(u)
16     k2 = F(u + 0.5 * dt * k1)
17     k3 = F(u + 0.5 * dt * k2)
18     k4 = F(u + dt * k3)
19
20     return u + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
21
22 def burgers_rk4(dx, t_final, v=0.2, start_x=0, end_x=1, return_history=False):
23     # Take ceil for num_nodes
24     num_nodes = int((end_x - start_x) / dx) + 1
25     x = np.linspace(start_x, end_x, num_nodes)
26
27     # The CFL condition for diffusion is more restrictive
28     # as for initial condition u = sin(2pi x), max|u| = 1
29     # Pick C_2 = 0.5 to be safe
30     dt = 0.5 * dx**2 / v
31     num_time_steps = int(np.ceil(t_final / dt))
32     # Adjust dt to fit exactly into t_final
33     dt = t_final / num_time_steps
34
35     u = initial_condition(x)
36
37     # Store history of u for Question 3.4
38     if return_history:

```

```
39     t = 0
40     u_history = [deepcopy(u)]
41     times = [t]
42
43     for _ in range(num_time_steps):
44         u = rk4_step(u, dt, dx, v)
45         if return_history:
46             t += dt
47             u_history.append(deepcopy(u))
48             times.append(t)
49
50     if return_history:
51         return x, u, np.array(times), np.array(u_history)
52
53     return x, u
54
55 if __name__ == "__main__":
56     dx = 1e-1
57     t_final = 1
58     x, u = burgers_rk4(dx, t_final)
59
60     # <Some matplotlib styling and enable LaTeX>
61
62     plt.plot(x, u, label=f'$u(x, {t_final})$')
63     plt.title("1D Burgers' Equation with RK4")
64     plt.xlabel("$x$")
65     plt.legend()
66     plt.savefig('./outputs_3/burgers_rk4.png', dpi=300)
67     plt.show()
```

The resulting plot at  $T = 1$  second was:





### Question 3.3

Compute  $\|u - u_h\|_{L^2}$  with a different refined mesh, and show your spatial convergence in a log-log plot.

*Solution.* The discrete  $L^2$  norm was used to compute our error:

$$\|u - u_h\|_{L^2} = \sqrt{\Delta x \sum_{i=1}^N (u_i - u_{h,i})^2}$$

Where  $N$  is the number of spatial nodes in the coarser mesh,  $u_i$  is the exact solution at node  $i$ , and  $u_{h,i}$  is the numerical solution at node  $i$ .

The following code was used to compute the  $L^2$  error and plot spatial convergence:

Listing 6: 3.3 Python

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def discrete_l2(u_num, u_ref, dx):
5     return np.sqrt(np.sum((u_num - u_ref)**2) * dx)
6
7 def spacial_convergence(dx_list, t_final, v=0.2):
8
9     # dx_exact will serve as our "exact" solution
10    # Use a finer dx than minimum dx in dx_list
11    dx_exact = min(dx_list) / 4
12    x_exact, u_exact = burgers_rk4(dx_exact, t_final, v)

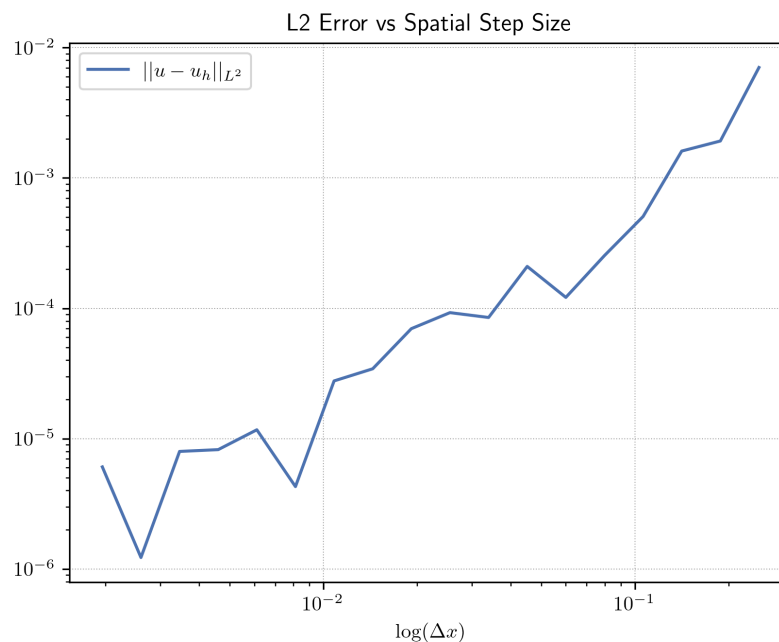
```

```

13
14 errors = []
15 for dx in dx_list:
16     x_approx, u_approx = burgers_rk4(dx, t_final, v)
17     # Interpolate exact solution to the current mesh of x_approx
18     u_exact_on_x_approx = np.interp(x_approx, x_exact, u_exact)
19     error = discrete_l2(u_approx, u_exact_on_x_approx, dx)
20     errors.append(error)
21 return dx_list, errors
22
23 if __name__ == "__main__":
24     # <Some matplotlib styling and enable LaTeX>
25     dx_list = np.logspace(-2, -9, 18, base=2)
26     dx, error = spacial_convergence(dx_list, t_final)
27     plt.loglog(dx, error, label='||u - u_h||_{L^2}')
28     plt.xlabel('$\\log(\\Delta x)$')
29     plt.title('L2 Error vs Spatial Step Size')
30     plt.legend()
31     plt.savefig('./outputs_3/burgers_rk4_error.png', dpi=300)
32     plt.show()

```

The plot generated for spatial convergence at  $T = 1$  was:



A few notes:

1.  $\Delta x = \frac{1}{2}$  has been excluded from the plot as it had favorable error due to the coarse mesh aligning well with the sine wave at  $T = 1$ .

2. The fluctuations in the error can be explained by additional interpolation error when mapping the exact solution to the coarser mesh and/or point 1 above.

Overall, we observe second-order spatial convergence as  $\Delta x$  decreases.

### Question 3.4

Plot your  $\|u - u_h\|_{L^2}$  vs time.

*Solution.* The following code was used to compute and plot the  $L^2$  error vs time:

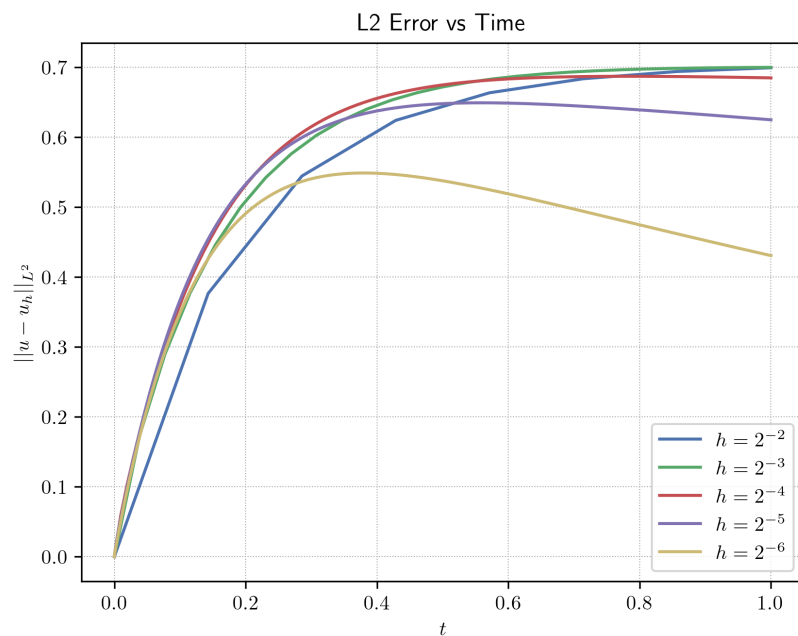
Listing 7: 3.4 Python

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def norm_vs_time(dx, t_final, x_exact, u_exact_history, v=0.2):
5     x_approx, _, times, u_history = burgers_rk4(dx, t_final, v, return_history=
6     True)
7
8     norms = []
9     for u_approx, u_exact in zip(u_history, u_exact_history):
10         u_exact_on_x_approx = np.interp(x_approx, x_exact, u_exact)
11         norm = discrete_l2(u_approx, u_exact_on_x_approx, dx)
12         norms.append(norm)
13     return times, norms
14
15 if __name__ == "__main__":
16     # <Some matplotlib styling and enable LaTeX>
17     dx_list = np.logspace(-2, -6, 5, base=2)
18
19     dx_exact = min(dx_list) / 4
20     x_exact, _, _, u_exact_history = burgers_rk4(dx_exact, t_final,
21     return_history=True)
22
23     for index, dx in enumerate(dx_list):
24         time, norms = norm_vs_time(dx, t_final, x_exact, u_exact_history)
25         plt.plot(time, norms, label=f'$h = 2^{{{-index-2}}}$')
26     plt.xlabel('$t$')
27     plt.ylabel(r'$\|u - u_h\|_{L^2}$')
28     plt.title('L2 Error vs Time')
29     plt.legend()
30     plt.savefig('./outputs_3/burgers_rk4_error_vs_time.png', dpi=300)
31     plt.show()

```

The plot generated for  $L^2$  error vs time was:



We can see that the  $L^2$  error generally increases with time and then stabilizes. The increase makes sense as numerical errors accumulate over time steps.

Finer meshes (smaller  $h$ ) consistently yield lower errors throughout the simulation at  $T = 1$ .