PHYS4840 Homework 4

Chase Worsley

March 24, 2025

GitHub Repo: link

Problem 0:

Done, see 3-4-25 and 3-6-25.

Problem 1:

First, let's find the first error of linear:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}\,, \quad h = \text{step size}$$

Taylor-series expansion around x:

$$\implies f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \,...$$
$$\implies f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \,...$$

$$\text{Plug back into linear Eq.:}$$

$$\begin{aligned} f'(x) &= \frac{(f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \,...) - (f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \,...)}{2h} \\ &= \frac{(\cancel{f(x)} + hf'(x) + \cancel{\frac{h^2}{2}f''(x)} + \,...) - (\cancel{f(x)} - hf'(x) + \cancel{\frac{h^2}{2}f''(x)} - \,...)}{2h} \\ &= \frac{2hf'(x) + \frac{h^3}{3}f'''(x) + \,...}{2h} \\ &= f'(x) + \frac{h^2}{6}f'''(x) + \,... \end{aligned}$$

Therefore, the linear central difference scheme is $O(h^2)$ accurate.

Now for the quadratic case (x^2):

$$\text{Standard quadratic: } f(x) = Ax^2 + Bx + C$$

$$f'(x) = a_1 f(x-h) + a_2 f(x) + a_3 f(x+h)$$

These a_n constants need to be able to differentiate for the standard quadratic above. Using the Taylor series we find that:

$$a_1 = \frac{-1}{2h}\,, \quad a_2 = 0\,, \quad a_3 = \frac{1}{2h}\,,$$

Plugging these back into the approximate derivative we get the same $f'(x)$ as the linear, meaning the error does not change.

Problem 2:

Done, see hw4.py, lines 3-92.

Problem 3:

1.

The matrix is invertible which means it is square and the determinant is non-zero. Also every row is linearly independent so there is a unique solution.

2.

Done, see hw4.py, lines 95-143

Problem 4:

Interpolation is used to estimate unknown values within the range of known data points by constructing a function that passes through these points. It's useful when you need to predict intermediate values, such as estimating temperature at a specific time between recorded hourly data.

Numerical differentiation estimates the derivative of a function using discrete data points, often through finite difference methods. This is applied when you need to determine the rate of change, like calculating velocity from position data recorded at specific time intervals.

TLDR: interpolation fills in data gaps, while numerical differentiation finds the slope or rate of change.

Problem 5:

Matrix A

$$\begin{vmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{vmatrix} = (4-\lambda)(3-\lambda) - (1)(2) = (\lambda - 2)(\lambda - 5) = 0$$

$$\implies \lambda_1 = 2 \quad \lambda_2 = 5$$

Matrix B

$$\begin{vmatrix} 1-\lambda & 2 & 3 \\ 0 & 1-\lambda & 4 \\ 0 & 0 & 1-\lambda \end{vmatrix} = (-\lambda{+}1)(-\lambda{+}1)(-\lambda{+}1){+}2{\cdot}4{\cdot}0{+}3{\cdot}0{\cdot}0{-}0{\cdot}(-\lambda{+}1){\cdot}3{-}0{\cdot}4{\cdot}(-\lambda{+}1){-}(-\lambda{+}1){\cdot}0{\cdot}2 = 0$$

$$\implies \lambda_1 = 1 \text{ , only solution :(}$$

Matrix C

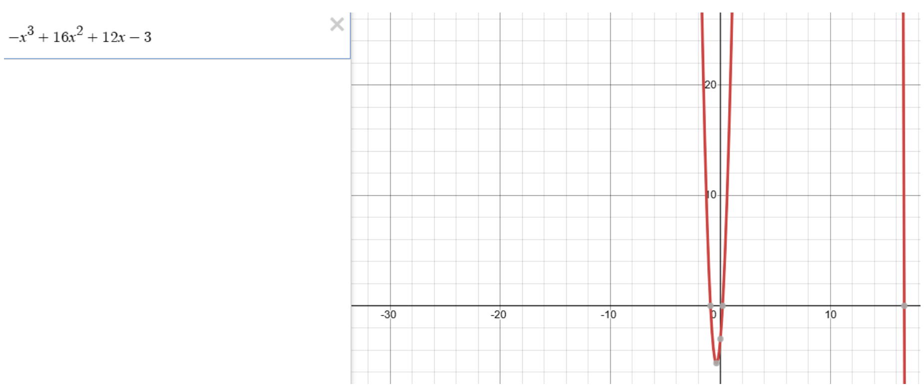
$$\begin{vmatrix} 1-\lambda & 2 & 3 \\ 4 & 5-\lambda & 6 \\ 7 & 8 & 9-\lambda \end{vmatrix} = -\lambda\left(\lambda + \frac{3\sqrt{33}-15}{2}\right)\left(\lambda - \frac{3\sqrt{33}+15}{2}\right) = 0$$

$$\implies \lambda_1 = 0\,, \quad \lambda_2 = -\frac{3\sqrt{33}+15}{2}\,, \quad \lambda_3 = \frac{3\sqrt{33}+15}{2}$$

Matrix D

$$\begin{vmatrix} 1-\lambda & 2 & 3 \\ 4 & 5-\lambda & 6 \\ 7 & 8 & 10-\lambda \end{vmatrix} = -\lambda^3 + 16\lambda^2 + 12\lambda - 3$$

I honestly couldn't figure out how to factor this and find the roots so I just plotted it in Desmos :/, sorry.



The roots are:

$$\lambda_1 = -0.90574\,, \quad \lambda_2 = 0.19825\,, \quad \lambda_3 = 16.70749$$

Code

```python
import numpy as np

# Problem 2
# Part a) ----------------------------------------------------------------------------------------------------------------
def trapezoidal_rule(f, a, b, N):
    """
    Approximates the integral using the trapezoidal rule with a loop.

    Parameters:
        f (function or array-like): A function, it's evaluated at N+1 points.

        a (float): Lower bound of integration.
        b (float): Upper bound of integration.
        N (int): Number of intervals (trapezoids).

    Returns:
        float: The approximated integral.
    """

    h = (b-a)/N

    integral = (1/2) * (f(a) + f(b)) * h  # Matches the first & last term in the sum

    # Loop through k=1 to N-1 to sum the middle terms
    for k in range(1, N):
        xk = a + k * h  # Compute x_k explicitly (matches the formula)
        integral += f(xk) * h  # Normal weight (multiplied by h directly)

    return integral

def adaptiveTrapezoidal(f, a, b, tol):
    n = 1
    integral_old = trapezoidal_rule(f, a, b, n)
    error = tol + 1

    while error > tol:
        n *= 2
        integral_new = trapezoidal_rule(f, a, b, n)
        error = np.abs(integral_new - integral_old) / 3
        integral_old = integral_new

    return integral_new, n

def func1(x):
    return (np.sin(np.sqrt(100*x)))**2

a = 0
b = 1
```

```python
tolerance = 1e-6

integral, intervals = adaptiveTrapezoidal(func1, a, b, tolerance)
print(f"Estimated integral: {integral}")
print(f"Number of intervals used: {intervals}")

# Part b) ------------------------------------------------------------------------------------------
print("Doing romberg now")

def adaptiveRomberg(f, a, b, tol):
    R = [[(b - a) * (f(a) + f(b)) / 2]]
    m = 1
    error = tol + 1

    while error > tol:
        m_old = m
        m *= 2
        h = (b - a) / m
        T_new = 0.5 * R[-1][0] + h * sum(f(a + (k + 0.5) * ((b - a) / m_old)) for k in range(m_old))
        R.append([T_new])

        i = len(R) - 1
        for j in range(1, i + 1):
            extrapolated = (4**j * R[i][j-1] - R[i-1][j-1]) / (4**j - 1)
            R[i].append(extrapolated)

        if i > 0:
            error = abs(R[i][i] - R[i-1][i-1])
        else:
            error = tol + 1

    return R[-1][-1], R

def rombergTable(R):
    for i, row in enumerate(R):
        print("R[{}]: {}".format(i, "\t".join(f"{val:.10f}" for val in row)))

a = 0
b = 1
tolerance = 1e-6

result, R = adaptiveRomberg(func1, a, b, tolerance)
rombergTable(R)
print(f"Integral: {result}")

# ------------------------------------------------------------------------------------------
# Problem 3
# Part 1)
# See pdf

```

```
# Part 2)
print("\n----------- Problem 3 -----------\n")
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
import myFuncLib as mfl

A = np.array([ [1, 0, 0, 0],\
         [0,1,1,-1],
         [0,2,4,0],
               [0,2,-1,2] ],float)
N = len(A)

L = np.array([[1.0 if i == j else 0.0 for j in range(N)] for i in range(N)])
U = A.copy()
for m in range(N):
    for i in range(m+1, N):
        L[i, m] = U[i, m] / U[m, m]
        U[i, :] -= L[i, m] * U[m, :]

print('The lower triangular matrix L is:\n', L)
print('The upper triangular matrix U is:\n', U)

vector = np.array([0,294.3,392.4,196.2],float)

Q, R = mfl.qr_decomposition(A)

print("Matrix Q:\n", Q)
print("Matrix R:\n", R)

# Part 3)
is_orthogonal = np.allclose(np.dot(Q.T, Q), np.eye(Q.shape[1]))
print("Is Q orthogonal? (Q^T Q = I):", is_orthogonal)

def isUpper(matrix):
    rows, cols = matrix.shape
    for i in range(1, rows):
        for j in range(i):
            if matrix[i, j] != 0:
                return False

    return True

print("Upper diagonality for Q?:", isUpper(Q))
print("Upper diagonality for R?:", isUpper(R))
```