MAT3110 — Assignment 1

Herman Scheele
(Dated: September 2025)

Here are the 4 plots generated for the assignment. The plots were made in **Python** with **matplotlib.pyplot**

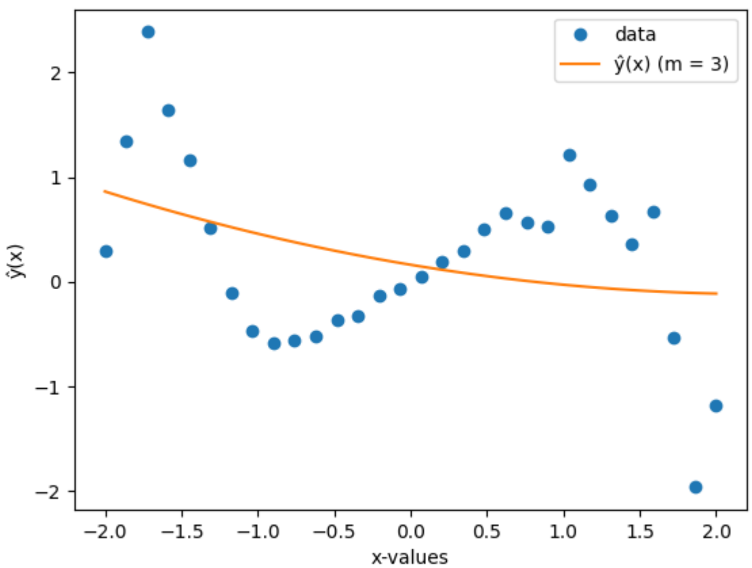

FIG. 1: caption

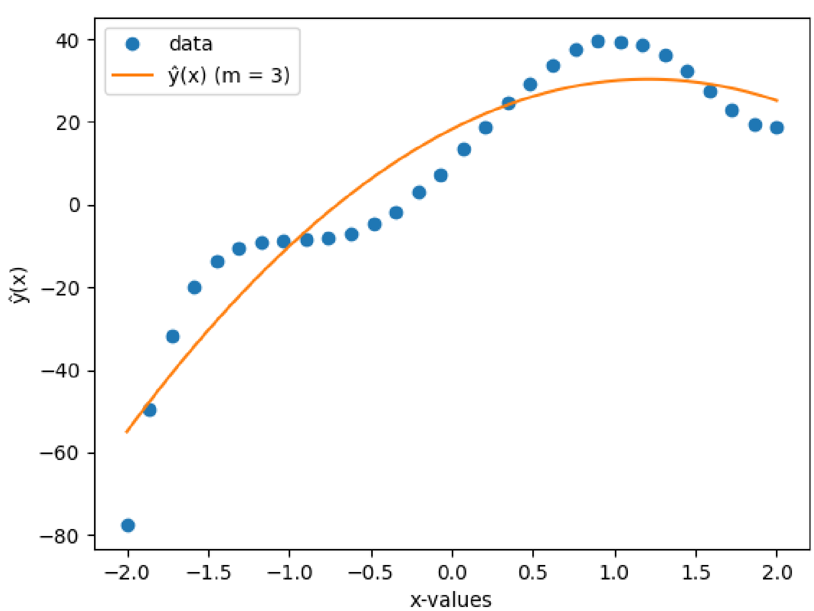

FIG. 2: caption

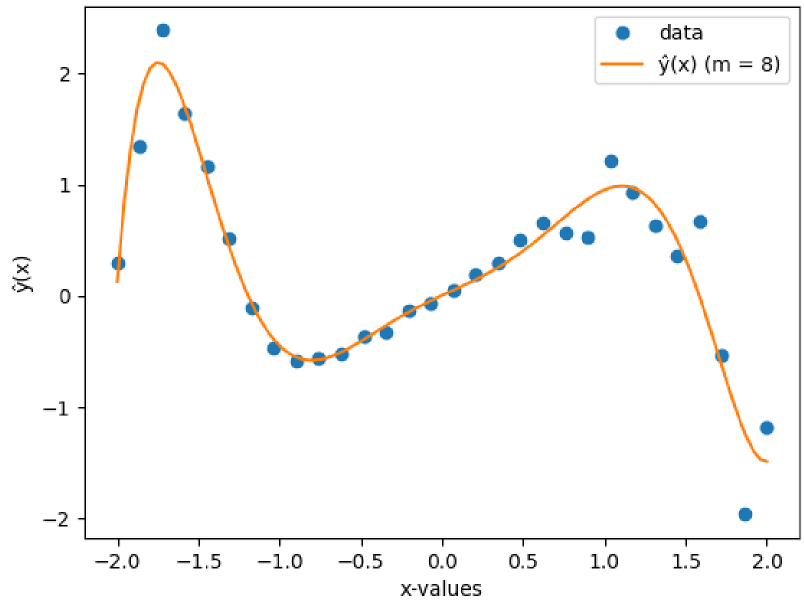

FIG. 3: caption

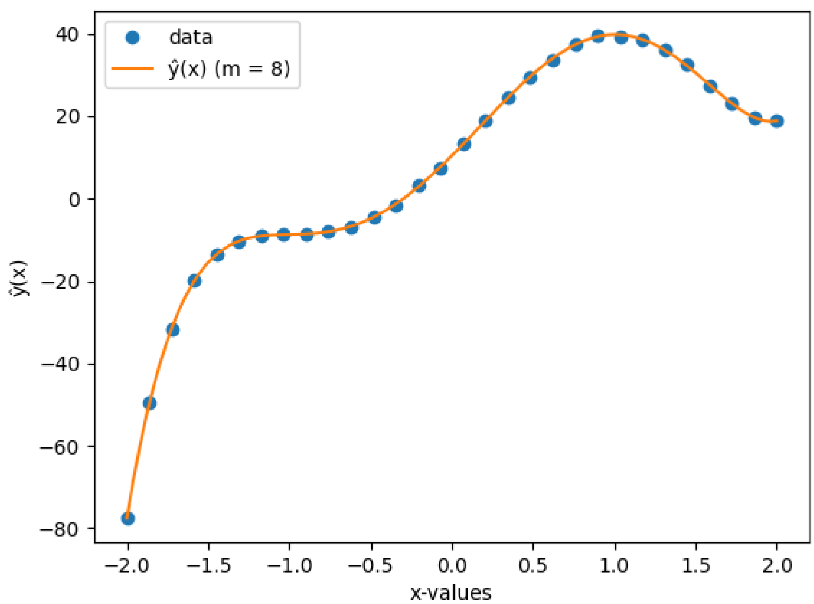

FIG. 4: caption

.

Here are the methods for the assignment implemented in **Python** with **Numpy**

```python
# Back-substitution
def back_sub_solver(A, b):
    n = len(A)
    x = np.zeros(n)
    x[n-1] = b[n-1] / A[n-1][n-1]

    for i in range(n - 2, -1, -1):
        s = 0
        for j in range(i + 1, n):
            s += A[i][j] * x[j]

        x[i] = (b[i] - s) / A[i][i]

    return x

# QR-Factorization
def QR(A):
    return np.linalg.qr(A)

# Generation of the Vandermonde matrix
def Vandermonde(x, m):

    n = len(x)
    A = np.zeros((n,m))
    A[:, 0] = np.ones(n)

    for i in range(1, m):
        ai = x ** i
        A[:, i] = ai

    return A

# Forward-substitution
def forward_sub_solver(A, b):
    n = len(A)
    x = np.zeros(n)
    x[0] = b[0] / A[0][0]

    for i in range(1, n):
        s = 0

        for j in range(i):
            s += A[i][j] * x[j]

        x[i] = (b[i] - s) / A[i][i]

    return x

# Cholesky-factorization
def cholesky_fact(A):

    Ai = A
    n = len(Ai)
    D = np.zeros((n, n))
    L = np.eye(n, n)

    for k in range(len(A)):

        lk = Ai[:, k] / Ai[:, k][k]
        outer_product = np.outer(lk, lk)

        Dkk = Ai[k][k]
        Ai = Ai - Dkk * outer_product

        D[k, k] = Dkk
        L[:, k] = lk

    return L, D
```

Here is the code for completing the task and generation of the 4 plots above.

```
from data import x, y, y2, n
from methods import back_sub_solver, QR, Vandermonde, forward_sub_solver, cholesky_fact
import numpy as np
import matplotlib.pyplot as plt


ms = [3, 8]
for m in ms:
    data = [y, y2]
    for yi in data:

        # ------------ Task 1 ------------ #

        A = Vandermonde(x, m)
        b = yi

        Q, R = QR(A)
        c = Q.T @ b

        R1 = R[:m, :m]
        c1 = c[:m]

        coeff = back_sub_solver(R1, c1)

        # ------------ Task 2 ------------ #

        B = A.T @ A
        e = A.T @ b

        L, D = cholesky_fact(B)
        R = L @ np.sqrt(D)

        u = back_sub_solver(R, e)
        coeff2 = forward_sub_solver(R.T, u)

        # ------------ Plot ------------ #

        # Plot of fitted pol. by QR-fact.
        xplot = np.linspace(x.min(), x.max(), 100)
        Aplot = Vandermonde(xplot, m)
        yfit = Aplot @ coeff
        plt.plot(x, yi, 'o', label="data")
        plt.plot(xplot, yfit, label=f"y  (x)  (m = {m})")
        plt.xlabel("x-values")
        plt.ylabel("y  (x)")
        plt.legend()
        plt.show()
```

Discussion

In Task 1 I solved the least squares problem using the QR factorization $A = QR$. After computing $Q^T b$, the reduced system $R_1 x = c_1$ was solved by back substitution to obtain the polynomial coefficients. In Task 2 I formed the normal equations $A^T Ax = A^T b$, set $B = A^T A$ and $e = A^T b$, and then solved $Bx = e$ using the Cholesky factorization $B = RR^T$. This required implementing both forward and back substitution.

The two approaches give the same solution, but their stability differs. The normal equations involve $A^T A$, which squares the condition number and can lead to larger numerical errors when m is large or A is ill-conditioned (as with high-degree Vandermonde matrices). QR is more stable, though a little more costly for memory.

For $m = 3$, both data sets are underfit and the curve does not capture the structure well. For $m = 8$, the fits are much closer to the data: the first data set shows the behaviour more accurately, and in the second data set the polynomial of degree 7 closely tracks the underlying 5th-degree polynomial. This illustrates the trade-off between too low and higher polynomial degree.