Università della Svizzera italiana

Faculty of Informatics

Introduction to Computational Science

Assignment6

Zhixiang Dai, Daniele del Pozzo

Spring Semester, May 29, 2025

1 Normal Equation I

Step 1: Derive the Normal Equations

Define the objective function:

$$f(x) = \|Ax - b\|_2^2 = (Ax - b)^\top (Ax - b)$$

Expanding:

$$f(x) = x^\top A^\top Ax - 2b^\top Ax + b^\top b$$

Take the gradient and set it to zero:

$$\nabla f(x) = 2A^\top Ax - 2A^\top b = 0 \Rightarrow A^\top Ax = A^\top b$$

This is the system of normal equations.

Step 2: Show that $A^\top A$ is Symmetric Positive Definite

- **Symmetry:** $A^\top A$ is symmetric since

$$(A^\top A)^\top = A^\top A$$

- **Positive Definite:** For any nonzero $x \in \mathbb{R}^n$,

$$x^\top A^\top Ax = \|Ax\|_2^2 > 0$$

because A has full column rank $\Rightarrow Ax \neq 0$ when $x \neq 0$.

Step 3: Uniqueness and Solution of the Least Squares Problem

Since $A^\top A$ is symmetric positive definite, it is invertible. Therefore, the normal equations have a unique solution:

$$x^* = (A^\top A)^{-1} A^\top b$$

Step 4: Final Soluton

Under the full column rank assumption, the matrix $A^\top A$ is **symmetric positive definite**, so the normal equations have a **unique** solution $x^* = (A^\top A)^{-1} A^\top b$, which is the unique minimizer of the least squares problem $\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2$.

2 Normal Equation II

Step 1: Formulate the System of Equations

For each data point (x_i, y_i):

$$\begin{aligned}
i = 0: &\quad a(-1)^2 + b(-1) + c = 1 \Rightarrow a - b + c = 1 \\
i = 1: &\quad a(0)^2 + b(0) + c = 0 \Rightarrow c = 0 \\
i = 2: &\quad a(1)^2 + b(1) + c = 1 \Rightarrow a + b + c = 1 \\
i = 3: &\quad a(2)^2 + b(2) + c = 0 \Rightarrow 4a + 2b + c = 0
\end{aligned}$$

Matrix form:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 4 & 2 & 1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Step 2: Compute Normal Equations

Compute $\mathbf{A}^T\mathbf{A}$:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 4 \\ -1 & 0 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 4 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 18 & 8 & 6 \\ 8 & 6 & 2 \\ 6 & 2 & 4 \end{bmatrix}$$

Compute $\mathbf{A}^T\mathbf{y}$:

$$\mathbf{A}^T\mathbf{y} = \begin{bmatrix} 1 & 0 & 1 & 4 \\ -1 & 0 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix}$$

Step 3: Solve the System

The normal equations:

$$\begin{cases} 18a + 8b + 6c = 2 \\ 8a + 6b + 2c = 0 \\ 6a + 2b + 4c = 2 \end{cases}$$

From $c = 0$ (from $i = 1$), simplify:

$$4a + 3b = 0$$
$$3a + b = 1$$

Solution:

$$b = 1 - 3a$$
$$4a + 3(1 - 3a) = 0 \Rightarrow a = \frac{3}{5}$$
$$b = 1 - 3\left(\frac{3}{5}\right) = -\frac{4}{5}$$
$$c = 0$$

Step 4: Final Solution

The best-fitting parabola is:

$$p(x) = \frac{3}{5}x^2 - \frac{4}{5}x$$

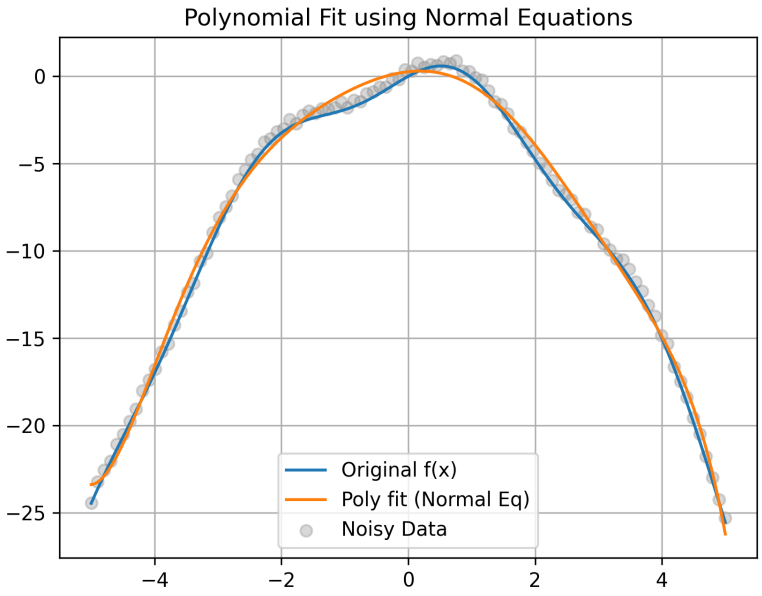
Verification

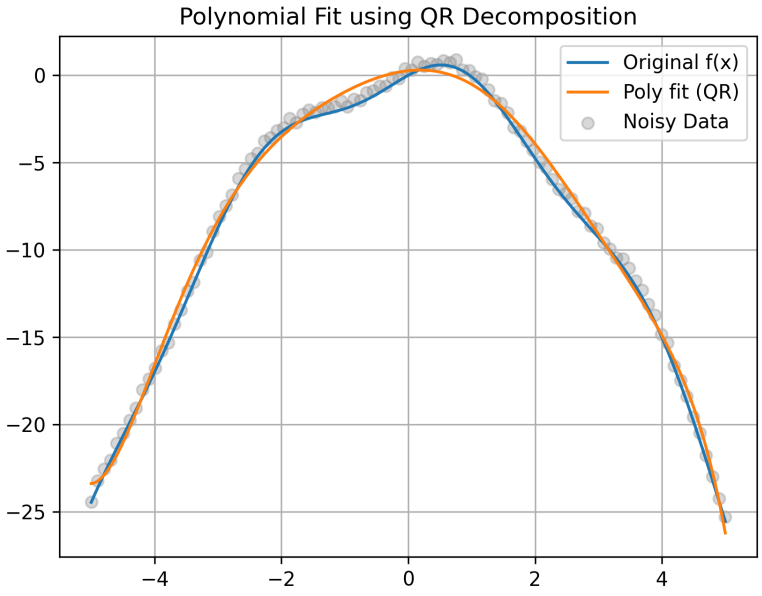
Residual:

$$\mathbf{y} - \mathbf{A}\mathbf{x} = \begin{bmatrix} -\frac{2}{5} \\ 0 \\ \frac{6}{5} \\ -\frac{4}{5} \end{bmatrix}, \quad \text{Sum of squared errors} = \frac{56}{25}$$

3 Normal Equation vs QR

3.1 Result





3.2 Code

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv, qr, cond
```

```
# Step (a): Generate data
n = 100
x = np.linspace(-5, 5, n)
f = lambda x: np.sin(2 * x) - x**2
y_true = f(x)
y = y_true + 0.5 * np.random.rand(len(x))  # add noise

# Step (b): Construct Vandermonde matrix for degree d = 8
d = 8
V = np.vander(x, d + 1, increasing=True)

# Step (c1): Solve using normal equations
V_T_V = V.T @ V
V_T_y = V.T @ y
a_normal = inv(V_T_V) @ V_T_y

# Step (c2): Solve using QR decomposition
Q, R = qr(V)
a_qr = inv(R) @ Q.T @ y

# Step (d): Print condition numbers
cond_VTV = cond(V_T_V)
cond_R = cond(R)
print("Condition number of V^T V (Normal Equation):", cond_VTV)
print("Condition number of R (QR Decomposition):", cond_R)

# Step (e): Plot results
x_plot = np.linspace(-5, 5, 500)
y_f = f(x_plot)

# Polynomial predictions
V_plot = np.vander(x_plot, d + 1, increasing=True)
y_poly_normal = V_plot @ a_normal
y_poly_qr = V_plot @ a_qr

# Plot for Normal Equations
plt.figure()
plt.plot(x_plot, y_f, label='Original f(x)')
plt.plot(x_plot, y_poly_normal, label='Poly fit (Normal Eq)')
plt.scatter(x, y, color='gray', alpha=0.3, label='Noisy Data')
plt.title('Polynomial Fit using Normal Equations')
plt.legend()
plt.grid(True)
plt.savefig("normal_eq_fit.png", dpi=300)

# Plot for QR Decomposition
plt.figure()
plt.plot(x_plot, y_f, label='Original f(x)')
plt.plot(x_plot, y_poly_qr, label='Poly fit (QR)')
plt.scatter(x, y, color='gray', alpha=0.3, label='Noisy Data')
plt.title('Polynomial Fit using QR Decomposition')
plt.legend()
plt.grid(True)
plt.savefig("qr_fit.png", dpi=300)

plt.show()
```

4 Polynomial Approximation of Nonlinear Model

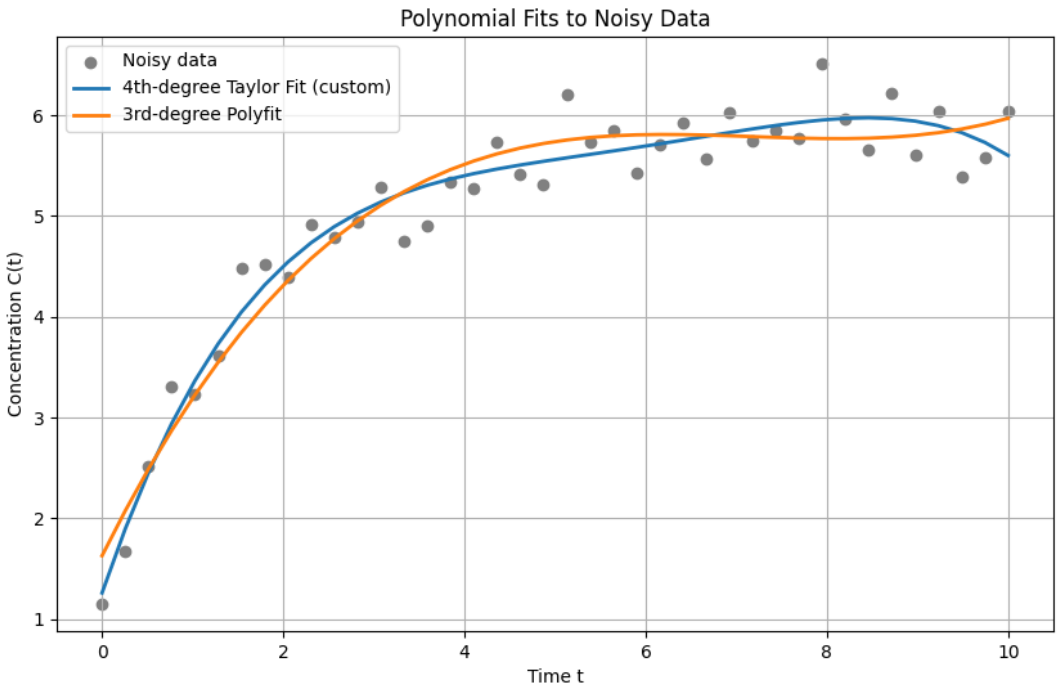
4.1 Result

4th-degree Taylor Polynomial Coefficients (custom least squares):
$\beta_0 = 1.2597,\ \beta_1 = 2.6410,\ \beta_2 = -0.6429,\ \beta_3 = 0.0725,\ \beta_4 = -0.0030$

3rd-degree Polynomial Coefficients (np.polyfit):
$\text{coeff}_0 = 0.0120,\ \text{coeff}_1 = -0.2571,\ \text{coeff}_2 = 1.8085,\ \text{coeff}_3 = 1.6284$

Model Comparison:
4th-degree Taylor Fit: RSS $= 2.8166$, $R^2 = 0.9511$
3rd-degree Polyfit: RSS $= 3.8141$, $R^2 = 0.9338$

Conclusion:
The 4th-degree Taylor model achieves a lower RSS and a higher R^2, indicating a better overall fit to the noisy exponential data.



4.2 Code

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.polynomial import Polynomial

# --- Parameters ---
A_true, k_true, B_true = 5, 0.6, 1
sigma = 0.3
n_points = 40
t = np.linspace(0, 10, n_points)
```

```
np.random.seed(42)

# --- Data Generation ---
C_true = A_true * (1 - np.exp(-k_true * t)) + B_true
noise = np.random.normal(0, sigma, size=n_points)
C_noisy = C_true + noise

# --- Custom least squares for 4th-degree Taylor polynomial ---
# Create design matrix for t^0 to t^4
X = np.vstack([np.ones_like(t), t, t**2, t**3, t**4]).T

# Solve least squares: beta = (X^T X)^-1 X^T y
XtX = X.T @ X
Xty = X.T @ C_noisy
beta = np.linalg.inv(XtX) @ Xty
C_fit_custom = X @ beta

# --- 3rd-degree polynomial using numpy.polyfit ---
coeffs_polyfit = np.polyfit(t, C_noisy, 3)
C_fit_polyfit = np.polyval(coeffs_polyfit, t)

# --- Plotting ---
plt.figure(figsize=(10, 6))
plt.scatter(t, C_noisy, color='gray', label='Noisy data')
plt.plot(t, C_fit_custom, label='4th-degree Taylor Fit (custom)', linewidth=2)
plt.plot(t, C_fit_polyfit, label='3rd-degree Polyfit', linewidth=2)
plt.title("Polynomial Fits to Noisy Data")
plt.xlabel("Time t")
plt.ylabel("Concentration C(t)")
plt.legend()
plt.grid(True)
plt.show()
plt.savefig("polynomial_fits.png")

# --- Residuals and Metrics ---
residuals_custom = C_noisy - C_fit_custom
residuals_polyfit = C_noisy - C_fit_polyfit
RSS_custom = np.sum(residuals_custom**2)
RSS_polyfit = np.sum(residuals_polyfit**2)
TSS = np.sum((C_noisy - np.mean(C_noisy))**2)
R2_custom = 1 - RSS_custom / TSS
R2_polyfit = 1 - RSS_polyfit / TSS

print("4th-degree Taylor Polynomial Coefficients (custom least squares):")
print(f"beta_0 = {beta[0]:.4f}, beta_1 = {beta[1]:.4f}, beta_2 = {beta[2]:.4f},
 ↪  beta_3 = {beta[3]:.4f}, beta_4 = {beta[4]:.4f}\n")

print("3rd-degree Polynomial Coefficients (np.polyfit):")
print(f"coeff_0 = {coeffs_polyfit[0]:.4f}, coeff_1 = {coeffs_polyfit[1]:.4f},
 ↪  coeff_2 = {coeffs_polyfit[2]:.4f}, coeff_3 = {coeffs_polyfit[3]:.4f}\n")

print("Model Comparison:")
print(f"4th-degree Taylor Fit -> RSS = {RSS_custom:.4f}, R^2 = {R2_custom:.4f}")
print(f"3rd-degree Polyfit    -> RSS = {RSS_polyfit:.4f}, R^2 =
 ↪  {R2_polyfit:.4f}")
```