Polynomial Regression and Visualization Using LAPACK and Cairo in C with Python Data Generation

Amaan Rahman

Introduction

In this project, a set of n sample points (x_i, y_i) are stored in a text file. The objective is to compute an optimal n-degree polynomial that best fits the given data in the least-squares sense. The computation of polynomial coefficients is performed in C using the high-performance numerical routines provided by **BLAS** and **LAPACK**, ensuring both efficiency and numerical stability.

The dataset is generated in Python, while the regression computation and visualization are done in C. The **Cairo Graphics Library** is used to plot the sample points along with the fitted polynomial curve, providing a clear visual representation of the model's accuracy.

Objective

To compute an **n-degree polynomial regression** that best fits a noisy dataset, using optimized numerical routines, and visualize the dataset along with the true polynomial curve.

Mathematical Formulation

For polynomial regression of degree n, the model is:

$$y = c_0 x^n + c_1 x^{n-1} + \cdots + c_{n-1} x + c_n.$$

In matrix form:

$$Y = X\beta + E,$$

where X is the Vandermonde matrix and E is the error vector.

Minimizing the squared error gives the Normal Equation:

$$X^T X\, \beta = X^T Y.$$

LAPACK solves this efficiently without computing the inverse.

OLS Formulation Using LAPACK

We represent the system in matrix form:

$$Y = X\beta + E, \quad E = Y - X\beta$$

Minimizing the sum of squared errors gives:

$$E^T E = (Y - X\beta)^T (Y - X\beta)$$

Differentiating and setting the gradient to zero:

$$X^T X\beta = X^T Y$$

Hence, the optimal parameter vector is:

$$\beta = (X^T X)^{-1} X^T Y$$

Rather than explicitly computing the inverse, LAPACK provides an efficient and numerically stable way to solve this system.

Implementation via LAPACK

Using LAPACK and BLAS, the steps are:

1. Compute $X^T X$ using `dgemm`.
2. Compute $X^T Y$ using `dgemv`.
3. Solve $(X^T X)\beta = X^T Y$ using `dposv` or `dgesv`.

Advantages of LAPACK Implementation

- **Efficiency:** Optimized for modern CPUs and parallel computation.
- **Numerical Stability:** Reduces rounding errors common in manual inversion.
- **Ease of Integration:** Directly linkable in C for high-performance computation.

Python Implementation for Generating Random Points

Part 1: Polynomial & Random Point Generation

```python
import numpy as np
import random
import sympy as sp
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def generate_points_and_polynomial():
    n = int(input("Enter the polynomial order n: "))
    x_vals = np.array([random.uniform(-10, 10) for _ in range(n)])
    y_vals = np.array([random.uniform(-10, 10) for _ in range(n)])
    coeffs = np.polyfit(x_vals, y_vals, n)
    x = sp.Symbol('x')
    poly_expr = sum(sp.Float(coeffs[i]) * x**(n - i) for i in range(n + 1)
        )

    print("\nGenerated Points (x, y):")
    for xi, yi in zip(x_vals, y_vals):
        print(f"({xi:.4f}, {yi:.4f})")

    print("\nPolynomial coefficients:")
    print(coeffs)

    print("\nPolynomial (Symbolic):")
    print(sp.expand(poly_expr))

    return n, coeffs, poly_expr
```

Part 2: Synthetic Data

```python
def generate_synthetic_data(poly_expr, x_range=(-5, 5), n_points=100, noise_std=1.0):
    x = np.linspace(x_range[0], x_range[1], n_points)
    y_true = np.array([poly_expr.subs('x', val) for val in x], dtype=float)
    y_noisy = y_true + np.random.normal(0, noise_std, size=len(x))
    return x, y_noisy, y_true

def build_curve_fit_model(order):
    def model(x, *params):
        return sum(params[i] * x**(order - i) for i in range(order + 1))
    return model

def main():
    n, coeffs, poly_expr = generate_points_and_polynomial()
    x_data, y_noisy, y_true = generate_synthetic_data(poly_expr, n_points=200, noise_std=2.0)
    model = build_curve_fit_model(n)
    initial_guess = np.ones(n + 1)
    popt, pcov = curve_fit(model, x_data, y_noisy, p0=initial_guess)
    y_fit = model(x_data, *popt)

    print("\nFitted Parameters:")
    print(popt)

    filename = "synthetic_polynomial_data.txt"
    with open(filename, "w") as f:
        f.write(f"Polynomial order: {n}\n")
        f.write(f"Number of data points: {len(x_data)}\n")
        f.write("x\ty_noisy\ty_true\n")
        for xi, yi_noisy, yi_true in zip(x_data, y_noisy, y_true):
            f.write(f"{xi:.6f}\t{yi_noisy:.6f}\t{yi_true:.6f}\n")

    print(f"\nData saved to '{filename}'.")

    plt.figure(figsize=(10, 6))
    plt.scatter(x_data, y_noisy, label="Noisy data", s=15)
    plt.plot(x_data, y_true, label="True polynomial", linewidth=2)
    plt.plot(x_data, y_fit, label="Fitted curve", linestyle='--', linewidth=2)
    plt.title(f"Polynomial Order {n}: True vs Fitted Curve")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.grid(True)
    plt.show()

main()
```

C Implementation for Solving Polynomial Coefficients Using LAPACK

C Program (Part 1): Reading Data and Memory Setup

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <Accelerate/Accelerate.h>

/* Safe malloc */
void *xmalloc(size_t s) {
    void *p = malloc(s);
    if (!p) { printf("Out of memory!\n"); exit(1); }
    return p;
}

/* Read data file generated by Python */
int read_data(const char *filename, int *order, int *N,
              double **x, double **y_noisy, double **y_true)
{
    FILE *f = fopen(filename, "r");
    if (!f) { printf("Error: cannot open file %s\n", filename); return -1;
        }
    char line[512];

    if (!fgets(line, sizeof(line), f)) return -1;
    sscanf(line, "Polynomial order: %d", order);
    if (!fgets(line, sizeof(line), f)) return -1;
    sscanf(line, "Number of data points: %d", N);
    fgets(line, sizeof(line), f); /* Skip header row */

    *x       = xmalloc(sizeof(double) * (*N));
    *y_noisy = xmalloc(sizeof(double) * (*N));
    *y_true  = xmalloc(sizeof(double) * (*N));

    for (int i = 0; i < *N; i++) {
        double xv, yn, yt;
        if (!fgets(line, sizeof(line), f)) { printf("Unexpected EOF at
            line %d\n", i); return -1; }
        if (sscanf(line, "%lf %lf %lf", &xv, &yn, &yt) != 3) { printf("Bad
             data line: %s\n", line); return -1; }
        (*x)[i] = xv; (*y_noisy)[i] = yn; (*y_true)[i] = yt;
    }
    fclose(f);
    return 0;
}
```

C Program (Part 2): Vandermonde Matrix Construction

```c
/* Build Vandermonde matrix (column-major for BLAS) */
void build_vandermonde(int N, int deg, const double *x, double *X)
{
    int P = deg + 1;
    for (int j = 0; j < P; j++) {
        int power = deg - j;
        for (int i = 0; i < N; i++) {
            X[j*N + i] = pow(x[i], power);
        }
    }
}
```

C Program (Part 3): LAPACK Solution and Coefficient Extraction

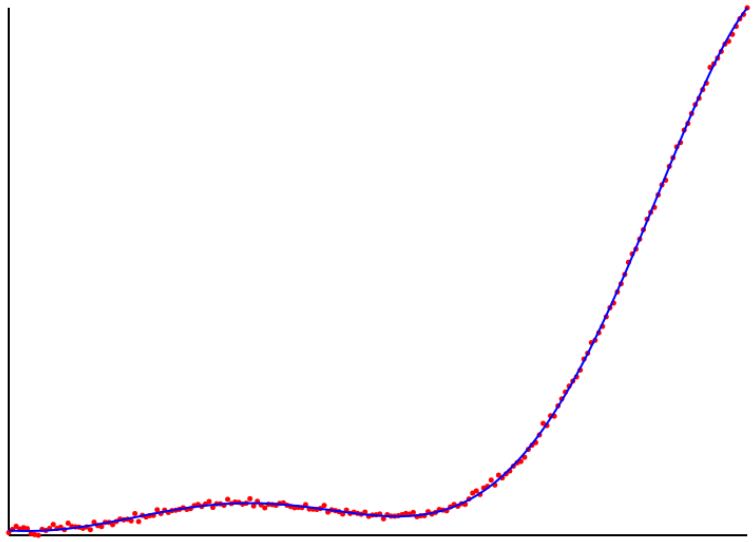
```c
int main(int argc, char **argv)
{
    const char *infile = "synthetic_polynomial_data.txt";
    if (argc > 1) infile = argv[1];

    int N, deg;
    double *x, *y_noisy, *y_true;

    printf("Reading: %s\n", infile);

    if (read_data(infile, &deg, &N, &x, &y_noisy, &y_true) != 0) {
        printf("File read error.\n");
        return 1;
    }

    printf("Loaded %d points, degree = %d\n", N, deg);

    int P = deg + 1;
    double *X   = xmalloc(sizeof(double) * N * P);
    double *XtX = xmalloc(sizeof(double) * P * P);
    double *Xty = xmalloc(sizeof(double) * P);

    build_vandermonde(N, deg, x, X);

    /* XtX = X^T X */
    cblas_dgemm(CblasColMajor, CblasTrans, CblasNoTrans,
                P, P, N, 1.0,
                X, N, X, N,
                0.0, XtX, P);

    /* Xty = X^T y */
    cblas_dgemv(CblasColMajor, CblasTrans,
                N, P, 1.0,
                X, N, y_noisy, 1,
                0.0, Xty, 1);

    /* Solve (XtX) beta = Xty with LAPACK */
    int n = P, nrhs = 1, lda = P, ldb = P, info;
    char uplo = 'U';

    dposv_(&uplo, &n, &nrhs, XtX, &lda, Xty, &ldb, &info);

    if (info != 0) {
        printf("Error: dposv failed with info=%d\n", info);
        return 1;
    }

    printf("\nOptimal Polynomial Coefficients:\n");
    for (int i = 0; i < P; i++) {
        printf("c[%d] = %.12f\n", i, Xty[i]);
    }

    FILE *out = fopen("optimal_coeffs.txt", "w");
    for (int i = 0; i < P; i++)
        fprintf(out, "%.12f\n", Xty[i]);
    fclose(out);
    printf("Saved coefficients to optimal_coeffs.txt\n");

    free(x); free(y_noisy); free(y_true);
```

```
60      free(X); free(XtX); free(Xty);
61      return 0;
62  }
```

Visualization using Cairo Library in C

After computing the regression line, visualization is done using the Cairo graphics library. Steps include:

- Initialize a Cairo surface and context using `cairo_create()`.
- Plot all n sample points as small circles/dots using `cairo_arc()` and `cairo_fill()`.
- Draw the polynomial curve using `cairo_move_to()` and `cairo_line_to()`.
- Label endpoints and annotate minimal distance using `cairo_show_text()`.
- Save the image to PNG using `cairo_surface_write_to_png()`.



Example plot for an 8th order polynomial using Cairo in C

Input/Output Format

Input:

- Text file containing n sample points (x_i, y_i).

Output:

- Optimal coefficients β minimizing the squared error.
- Visualization of data points and best-fit polynomial curve.