Numerical Methods: Assignment #4

Hong Chenhui
drredthered.github.io

Zhejiang University — April 8, 2024

Info: Gauss-Newton method was never introduced in the formal class. It was intended as a supplement on the regression matter. Complains on this is written in the following section.

1 Problem

- **Question** The following data define the sea-level concentration of dissolved oxygen for fresh water as a function of temperature:

$T/°C$	$o/mg \cdot L^{-1}$
0	14.621
8	11.843
16	9.870
24	8.418
32	7.305
40	6.413

Estimate $o(27)$ using interpolations as well as regressions. Note that the exact result is 7.986 mg/L.

1.1 Theoretical viewpoint

Question

Nothing to explain on this matter. Some typical methods one could choose are

Algorithm 1: Lagrange Polynomial

Input: Data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$
Output: Lagrange polynomial $P(x)$

```
for i = 0 to n do
    L_i(x) ← 1;
    for j = 0 to n do
        if i ≠ j then
            L_i(x) ← L_i(x) · (x − x_j)/(x_i − x_j);
        end
    end
end
P(x) ← 0;
for i = 0 to n do
    P(x) ← P(x) + y_i · L_i(x);
end
return P(x);
```

Question

Algorithm 2: Cubic Spline Interpolation

Data: Data points (x_i, y_i) for $i = 0, 1, \ldots, n$
Result: Cubic spline interpolant $S(x)$

Compute the coefficients a_i, b_i, c_i, d_i for $i = 0, 1, \ldots, n-1$;
for $i = 0$ *to* $n-1$ **do**
  $h_i = x_{i+1} - x_i$;
  $a_i = y_i$;
  $b_i = \frac{y_{i+1}-y_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1})$;
  $d_i = \frac{c_{i+1}-c_i}{3h_i}$;
end
Solve the tridiagonal system of equations for c_i;
Set up the tridiagonal system $A\mathbf{c} = \mathbf{d}$, where A is a tridiagonal matrix;
for $i = 1$ *to* $n-2$ **do**
  $A_{i,i-1} = h_{i-1}$;
  $A_{i,i} = 2(h_{i-1} + h_i)$;
  $A_{i,i+1} = h_i$;
end
Solve $A\mathbf{c} = \mathbf{d}$ for $\mathbf{c}$ using a tridiagonal solver;

Result: Cubic spline interpolant $S(x)$
for $i = 0$ *to* $n-1$ **do**
  $S(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$ for $x \in [x_i, x_{i+1}]$;
end

Algorithm 3: Polynomial Regression

Data: Data points (x_i, y_i) for $i = 1, 2, \ldots, n$ and degree d
Result: Coefficients $c_0, c_1, \ldots, c_d$ of the polynomial regression model

Initialize the design matrix $\mathbf{X}$ and response vector $\mathbf{y}$;
for $i = 1$ *to* n **do**
  Construct the row vector $\mathbf{x}_i = [1, x_i, x_i^2, \ldots, x_i^d]$;
  Append $\mathbf{x}_i$ to the design matrix $\mathbf{X}$;
  Append y_i to the response vector $\mathbf{y}$;
end
Compute the coefficient vector $\mathbf{c} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ using least squares;

Compare to see the difference.

2 Implementation

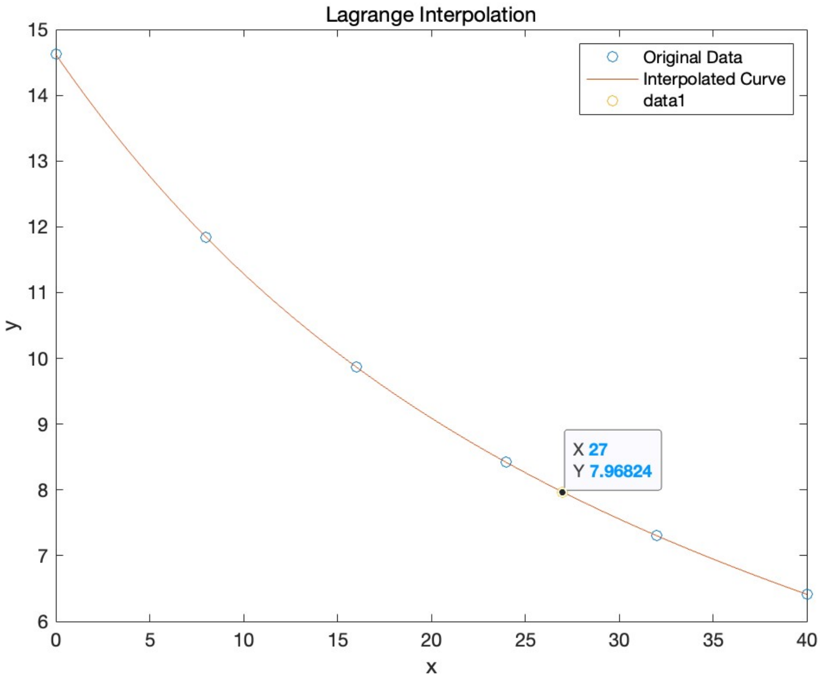

Fig 1. Lagrange Interpolation

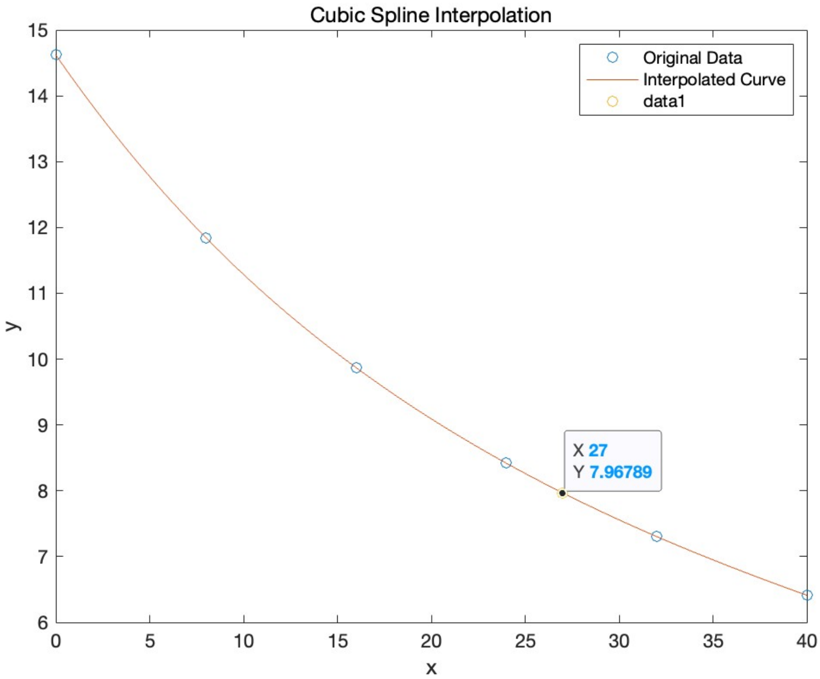

Fig 2.Cubic Spline Interpolation

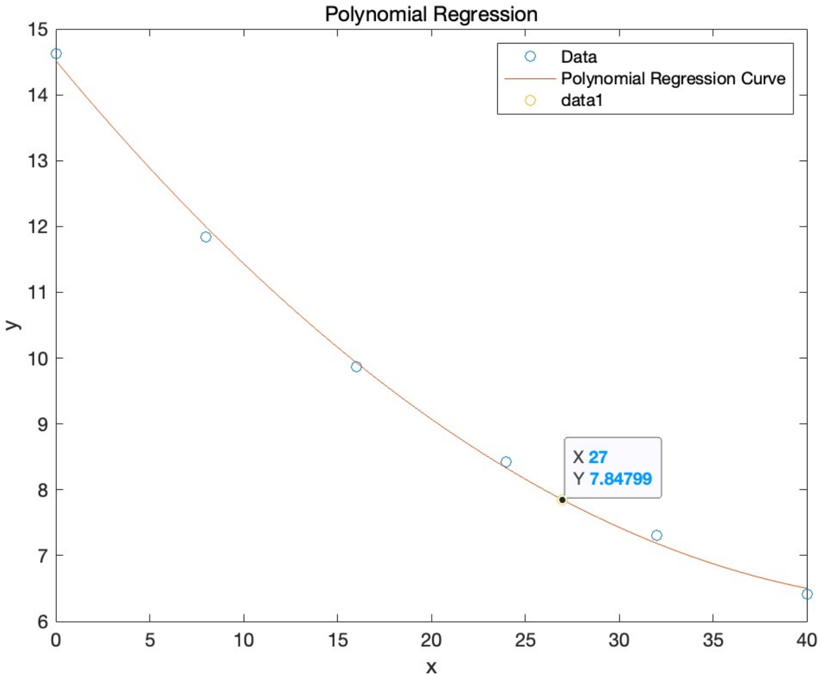

Fig 3.Polynomial Regression

Method	Value	Discrepancy%
Lagrange	7.968	0.225
CubicSpline	7.968	0.225
Polynomial	7.848	1.728

3 Analysis

3.1 Interpolation is better than regression

As shown on the graph above, the interpolation methods are more accurate than the regression method.

Interpolation methods often capture the local behavior of the data more accurately because they use information from nearby data points to estimate values at intermediate points. This can lead to more accurate predictions within the range of the data.

To put more specifically, the Interpolation assumes you have the function of interest – that you are observing the quantity of interest without noise. Regression assumes there is noise and tries to model the conditional mean —— which is not the case in this question, as the data is given pretty much without noise(we're not interested in given the exact correlation between the two vector. It is very different from performing a physical experiment where one struggle to find the relation between the two).

3.2 So how can you tell?

Say we're trying a different approach. Instead of using polynomial regression, we're assuming that a non-linear exponential form is closer to its physical essence. Such non-linear regression could be performed using Gauss-Newton Method as following.

- Requirements

$$\min_{\beta} \sum_{i=1}^{n} (y_i - f(x_i, \beta))^2 \tag{1}$$

- Expected function form

$$f(x) = \beta_1 e^{\beta_2 x} \tag{2}$$

Algorithm 4: Gauss-Newton Method

Data: Initial guess $\beta^{(0)}$, tolerance ϵ, maximum number of iterations N
Result: Optimal parameter estimate $\hat{\beta}$

Set iteration counter $k = 0$;
while $k < N$ **do**
  Compute the Jacobian matrix $J(\beta^{(k)})$;
  Compute the residual vector $r(\beta^{(k)})$;
  Solve the normal equations $J^T J \Delta\beta = -J^T r$ for $\Delta\beta$;
  Update the parameter estimate: $\beta^{(k+1)} = \beta^{(k)} + \Delta\beta$;
  if $\|\Delta\beta\| < \epsilon$ **then**
    break;
  end
  Increment iteration counter: $k = k + 1$;
end

resulted in

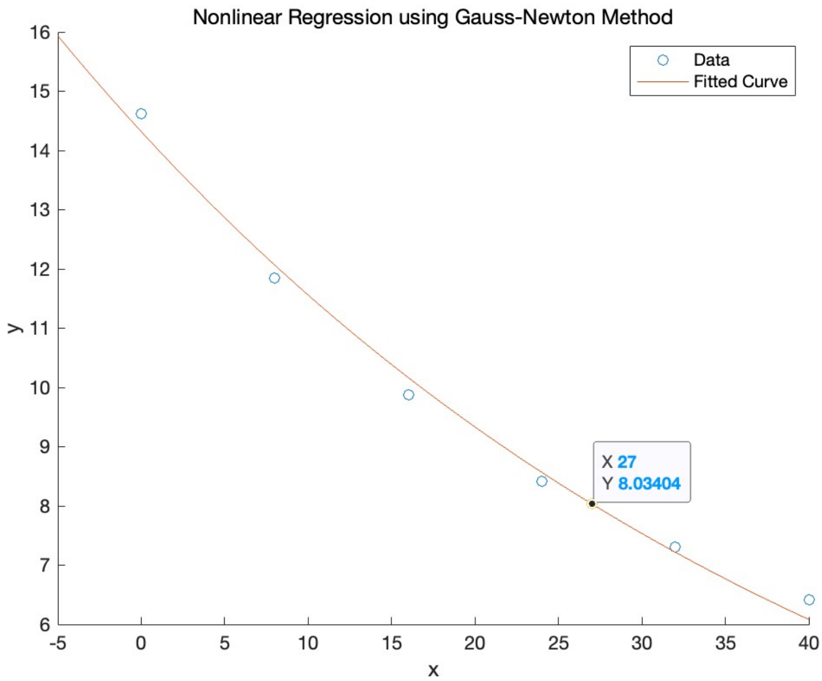

Fig 4. Non-linear Regression

Value	Rel.Error%	β_1	β_2	R^2
8.034	0.601	14.318	-0.02	0.9922

with 95% confidence interval of β_1 and β_2 being

β_1	β_2
$(13.6260, 15.0102)$	$(-0.0242, -0.0187)$

Which is still pretty terrible although better than the original polynomial guesses.

Another reason not using the Gauss-Newton method is the off-putting convergence result. The initial value of $\beta^{(0)}$ vector must be extremely close to the actual result to reach a satisfying converging rate, or else it would cause redundancy or not even converging at all. For instance, this vector would fly away.

$$\beta^{(0)} = \begin{bmatrix} 14 \\ -1 \end{bmatrix} \tag{3}$$

4 Codes

//functions

lagrange_interp.m

```
function lagrange_interp(x, y, x_val)
    n = length(x);
    L = ones(n,length(x_val));

    for i = 1:n
        for j = 1:n
            if i ~= j
                L(i,:) = L(i,:) .* (x_val - x(j)) / (x(i) - x(j));
            end
        end
    end

    interpolated_y = zeros(1,length(x_val));
    for i = 1:n
        interpolated_y = interpolated_y + y(i) * L(i,:);
    end

    disp('Interpolated values:');
    disp(interpolated_y);

    % Plot the interpolated polynomial
    plot(x, y, 'o', x_val, interpolated_y);
    legend('Original Data', 'Interpolated Curve');
    xlabel('x');
    ylabel('y');
    title('Lagrange Interpolation');

    hold on
    %specifically for chap 4%
    plot(x_val(271),interpolated_y(271),'o')
    disp('Exact:');
    disp(interpolated_y(271));
    %delete afterwards%
end
```

spline.m

```
function spline(x,y,x_val)

        y_interp = interp1(x, y, x_val, 'spline');

        % Plot original data and interpolated curve
        plot(x, y, 'o', x_val, y_interp, '-');
        legend('Original Data', 'Interpolated Curve');
        xlabel('x');
        ylabel('y');
        title('Cubic Spline Interpolation');
        hold on
        plot(x_val(271),y_interp(271),'o');

        disp("Exact:");
        disp(y_interp(271));
end
```

polyfit.m

```
function Polyfit(x,y,x_val)

        % Degree of the polynomial
        degree = 2;

        % Perform polynomial regression
        coefficients = polyfit(x, y, degree);

        % Generate polynomial values
        y_predicted = polyval(coefficients, x_val);

        % Plot the data and polynomial regression curve
        plot(x, y, 'o', x_val, y_predicted, '-');
        xlabel('x');
        ylabel('y');
        title('Polynomial Regression');
        legend('Data', 'Polynomial Regression Curve');

        hold on
        plot(x_val(271),y_predicted(271),'o');

        disp("Exact:");
        disp(y_predicted(271));
end
```

//script
DissolvedOxygen.m

```
        clear;
        clc;

        x = [0,8,16,24,32,40];
        y = [14.621,11.843,9.870,8.418,7.305,6.413];

        x_val = 0:0.1:40;

        figure(1);
        lagrange_interp(x, y, x_val);

        figure(2);
        spline(x,y,x_val);

        figure(3);
        Polyfit(x,y,x_val);
```

Gauss_Newton.m

```
clear;
clc;

x = [0,8,16,24,32,40];
y = [14.621,11.843,9.870,8.418,7.305,6.413];

% Plot the sample data
figure;
scatter(x, y);
hold on;

% Gauss-Newton Method
```

```
beta0 = [14; -0.1]; % Initial guess for parameters
beta = beta0;
max_iters = 100; % Maximum number of iterations
tol = 1e-6; % Tolerance for convergence

for iter = 1:max_iters
    % Calculate the Jacobian matrix
    J = zeros(length(x), length(beta));
    for i = 1:length(x)
        J(i, 1) = exp(beta(2) * x(i));
        J(i, 2) = beta(1) * x(i) * exp(beta(2) * x(i));
    end

    % Calculate the residuals
    residuals = y - model(x, beta);

    % Compute the step
    step = (J' * J) \ (J' * residuals');

    % Update the parameters
    beta = beta + step;

    % Check for convergence
    if norm(step) < tol
        break;
    end
end

% Plot the fitted curve
f = @(x) beta(1) * exp(beta(2) * x);
fplot(f);
plot(27,f(27),'o');
legend('Data', 'Fitted Curve');
xlabel('x');
ylabel('y');
title('Nonlinear Regression using Gauss-Newton Method');

% Define your nonlinear model function
function y = model(x, beta)
    % Example: y = beta(1) * exp(beta(2) * x)
    y = beta(1) * exp(beta(2) * x);
end
```