Project report:
Gauss-Newton optimization method

Nicolas Munke Cilano
Vidar Gimbringer
Artis Vijups

Supervisor: Stefan Diehl

December 14, 2024

1 Introduction

In this project, we mainly investigate the problem of fitting the function

$$\varphi(\boldsymbol{x}; t) = x_1 e^{-x_2 t} + x_3 e^{-x_4 t},$$

where $\boldsymbol{x} = (x_1, x_2, x_3, x_4)^T$ are parameters, to a given set of data points $(t_i, y_i), i = 1, \ldots, m$.

In particular, we seek to minimize the sum of the squared distances from each data point (t_i, y_i) to the point $(t_i, \varphi(\boldsymbol{x}; t_i))$. If we denote the distance by $r_i(\boldsymbol{x}) = \varphi(\boldsymbol{x}; t_i) - y_i$, then our goal is to

$$\underset{\boldsymbol{x} \in \mathbb{R}^4}{\text{minimize}}\ f(\boldsymbol{x}) \quad \text{where} \quad f(\boldsymbol{x}) = \sum_{i=1}^{m} r_i(\boldsymbol{x})^2.$$

To solve this minimization problem, we use the Gauss-Newton method.

2 Methods

2.1 Gauss-Newton method

Let $r(\boldsymbol{x}) = (r_1(\boldsymbol{x}), \dots, r_m(\boldsymbol{x}))^T$. The method is set up on the idea that

$$r(\boldsymbol{x} + \boldsymbol{\delta}) \approx r(\boldsymbol{x}) + J(\boldsymbol{x})\boldsymbol{\delta},$$

where $J(\boldsymbol{x})$ is the Jacobian matrix of $r(\boldsymbol{x})$ and $\boldsymbol{\delta}$ is an increment vector, also a direction vector.

This can be thought of as an extension to more dimensions of the strategy of moving along the tangent line to estimate a nearby value.

We then observe that

$$\begin{aligned} f(\boldsymbol{x} + \boldsymbol{\delta}) &= \sum_{i=1}^{m} r_i(\boldsymbol{x} + \boldsymbol{\delta})^2 \\ &= r(\boldsymbol{x} + \boldsymbol{\delta})^T r(\boldsymbol{x} + \boldsymbol{\delta}) \\ &\approx (r(\boldsymbol{x}) + J(\boldsymbol{x})\boldsymbol{\delta})^T (r(\boldsymbol{x}) + J(\boldsymbol{x})\boldsymbol{\delta}). \end{aligned}$$

Since we want the output of f to be the minimum possible, the gradient of this approximation should be 0. Writing that as an equation and simplifying, we end up with

$$J(\boldsymbol{x})^T J(\boldsymbol{x})\boldsymbol{\delta} = -J(\boldsymbol{x})^T r(\boldsymbol{x}).$$

This presents the following iterative algorithm:

1. Solve the linear system $J(\boldsymbol{x})^T J(\boldsymbol{x})\boldsymbol{\delta} = -J(\boldsymbol{x})^T r(\boldsymbol{x})$ for $\boldsymbol{\delta}$.
2. Determine an optimal step length λ for direction $\boldsymbol{\delta}$ using a line search algorithm.
3. Update $\boldsymbol{x}$ to $\boldsymbol{x} + \lambda\boldsymbol{\delta}$.

We repeat these actions until the step $\lambda\boldsymbol{\delta}$ is smaller than our chosen tolerance.

2.2 Line search algorithm

For the line search mentioned in the second step of the Gauss-Newton iteration algorithm, we use Armijo's rule on $F(\lambda) = f(\boldsymbol{x} + \lambda\boldsymbol{\delta})$.

Let $T(\lambda) = F(0) + \varepsilon F'(0)\lambda$ be a straight line through $(0, F(0))$ with less negative slope than the point's tangent, so $0 < \varepsilon < 1$.

Armijo's rule is made up of two (upper and lower) conditions, which are

$$F(\lambda) \leq T(\lambda) \quad \text{and} \quad F(\alpha\lambda) \geq T(\alpha\lambda) \text{ for fixed } \alpha > 1.$$

This rule ensures λ will be in an *interval* of points where F is substantially smaller. Computationally, this is faster than looking for a perfect choice for λ.

So that λ satisfies Armijo's rule, we choose it as follows:

1. Make an initial guess for λ.
2. Repeatedly scale λ up by α until it satisfies the lower condition.
3. Repeatedly scale λ down by α until it satisfies the upper condition.

3 Project work

3.1 Responsibilities

3.2 Structure

The project is stored on GitHub as a repository. It is made up of:

- phi1.m, phi2.m, data1.m, data2.m, grad.m as provided,
- gaussnewton.m, the implementation of the Gauss-Newton method,
- line_search.m, the line search algorithm based on Armijo's rule,
- script.m, the main script containing tests and tasks,
- report.tex and report.pdf, forming this report,
- metadata files.

3.3 Results

We fit the two functions from phi1.m and phi2.m,

$$\varphi_1(\boldsymbol{x};t) = x_1 e^{-x_2 t}, \quad \boldsymbol{x} = (x_1, x_2)^T$$

and

$$\varphi_2(\boldsymbol{x};t) = \varphi(\boldsymbol{x};t) = x_1 e^{-x_2 t} + x_3 e^{-x_4 t}, \quad \boldsymbol{x} = (x_1, x_2, x_3, x_4)^T,$$

to two sets of data points, `d1` and `d2` from data1.m and data2.m.

The results are:

Case	Optimal point $\boldsymbol{x}$	$\|\nabla f(\boldsymbol{x})\|_2$	$\|r(\boldsymbol{x})\|_\infty$
`d1`, φ_1	$(10.8108, 2.4786)^T$	$3.7450 \cdot 10^{-4}$	1.6287
`d2`, φ_1	$(12.9789, 1.7861)^T$	$8.0031 \cdot 10^{-2}$	1.0397
`d1`, φ_2	$(6.3445, 10.5866, 6.0959, 1.4003)^T$	$4.0651 \cdot 10^{-5}$	0.4334
`d2`, φ_2	$(4.1741, 0.8747, 9.7390, 2.9208)^T$	$9.5220 \cdot 10^{-3}$	0.1182

Appendix

gaussnewton.m

```
function [x, N_eval, N_iter, normg] = gaussnewton(phi, t, y, x0, tol, printout, plotout)
    tic; % Start timing

    % Inputs:
    % phi      - Function handle for the model function phi(x, t)
    % t        - Vector of independent variable data
    % y        - Vector of dependent variable data (observations)
    % x0       - Initial guess for the parameter vector x
    % tol      - Tolerance for stopping criteria
    % printout - Printing intermediate results (1: print, 0: no print)
    % plotout  - Plotting results (1: plot, 0: no plot)

    % Outputs:
    % x        - Final parameter estimate
    % N_eval   - Number of function evaluations
    % N_iter   - Number of iterations performed
    % normg    - Norm of the gradient at the final parameter estimate

    % 1. Initialize variables
    x = x0;         % Start with the initial guess
    N_eval = 0;     % Initialize function evaluation counter
    N_iter = 0;     % Initialize iteration counter
    max_iter = 100; % Set a maximum number of iterations to prevent infinite loops
    epsilon = 1e-6; % Regularization parameter for stability

    % 2. Define residual and objective function
    r = @(x) phi(x, t) - y; % Residual function: r(x) = phi(x, t) - y
    f = @(x) sum(r(x).^2);  % Objective function: f(x) = ||r(x)||^2

    % 3. Gauss-Newton loop
    while N_iter < max_iter
        N_iter = N_iter + 1; % Increment the iteration counter

        % 3.1. Evaluate the residual vector r(x)
        residual = r(x);
        N_eval = N_eval + 1;  % Increment function evaluation count

        % 3.2. Compute the Jacobian matrix J numerically
        m = length(residual); % Number of data points
        n = length(x);        % Number of parameters
        J = zeros(m, n);      % Initialize Jacobian matrix

        % Compute the Jacobian using grad.m
        for i = 1:m
            % Define scalar function for i-th residual
            scalar_ri = @(xi) phi(xi, t(i)) - y(i); % Residual for the i-th data point
            J(i, :) = grad(scalar_ri, x)';          % Use grad.m to compute its gradient
        end

        % 3.3. Compute search direction
        % Gauss-Newton direction - No regularization term:
        % d = -(J' * J) \ (J' * residual);
        % Gauss-Newton direction - With regularization term for numerical stability:
        d = -((J' * J) + epsilon * eye(size(J, 2))) \ (J' * residual);

        % 3.4. Perform line search to determine optimal step length
        fprintf('Gauss-Newton Iteration %d: Starting line search...\n', N_iter);
        F = @(lambda) f(x + lambda * d); % Define F(lambda)
        [lambda, N_eval] = line_search(F, 1, N_eval); % Call line search with init guess
                                                      % for lambda and update N_eval

        % 3.5. Update parameters
        x = x + lambda * d;

        % 3.6. Check for convergence
        if norm(lambda * d) < tol
            % Compute the maximum absolute residual
            max_residual = max(abs(residual));

            fprintf(['Gauss-Newton Iteration %d: Converged with tolerance %.4e' ...
                'and max(abs(r)) %.4e.\n'], N_iter, tol, max_residual);
            break; % Exit loop if update step is below the tolerance
        end

        % Print intermediate results if printout flag is set
        if printout
            % Compute the maximum absolute residual
            max_residual = max(abs(residual));

            % Compute the gradient norm
            normg = norm(2 * J' * residual);

            % Print the iteration details
            fprintf('Gauss-Newton Iteration %d:\n', N_iter);
            fprintf('x = [%.4f, %.4f, %.4f, %.4f]\n', x);
            fprintf('max(abs(r)) = %.4e, norm(grad) = %.4e, lambda = %.4e\n', ...
                max_residual, normg, lambda);
        end
    end

    % Print final results
    normg = norm(2 * J' * residual); % Final gradient norm
    fprintf('\nFinal Results:\n');
    fprintf('x = [%.4f, %.4f, %.4f, %.4f]\n', x);
    fprintf('Final norm(grad) = %.4e\n', normg);
    fprintf('Total iterations = %d\n', N_iter); % Number of Gauss-Newton iterations
    fprintf('Total function evaluations = %d\n', N_eval); % Evaluations in Gauss-Newton
                                                          % and line search

    elapsed_time = toc; % Stop timing
    fprintf('Total elapsed time: %.4f seconds\n', elapsed_time); % Display elapsed time

    % Plot results if requested
    if plotout
        figure; % Create a new figure
        plot(t, y, 'ro', 'MarkerSize', 6, 'DisplayName', 'Data'); % Plot data points
        hold on;
        % Plot fitted curve:
        plot(t, phi(x, t), 'b-', 'LineWidth', 1.5, 'DisplayName', 'Fitted Curve');
        legend show; % Add a legend
        title('Gauss-Newton Fit');
        xlabel('t'); % Label x-axis
        ylabel('y'); % Label y-axis
        grid on; % Add grid -> better visualization
    end
end
```

line_search.m

```
function [lambda, N_eval] = line_search(F, lambda0, N_eval)

    % Inputs:
    % F       - Function handle for F(lambda)
    % lambda0 - Initial guess for lambda
    % N_eval  - Function evaluation counter to be updated

    % Outputs:
    % lambda  - Step length that minimizes F(lambda)
    % N_eval  - Updated function evaluation counter

    % Parameters
    alpha = 2;         % Alpha from Alg 3, p.53
    lambda = lambda0;  % Initial lambda
    F0 = F(0);         % Initial F(lambda) value
    gf = grad(F,0);    % F'(0)
    ep = 0.1;          % Epsilon from Alg 3, p.53

    % Increment function evaluations for F(0)
    N_eval = N_eval + 1;

    % Increase lambda while sufficient reduction is not met
    while F(alpha*lambda) < F0 + ep * gf * alpha * lambda
        lambda = alpha * lambda;
        N_eval = N_eval + 1;
        fprintf(['  Line Search Iteration: lambda = %.4e, ' ...
            'F(lambda) = %.4e\n'], lambda, F(lambda));
    end

    % Decrease lambda while F(lambda) is still too large
    while F(lambda) > F0 + ep * gf * lambda
        lambda = lambda / alpha;
        N_eval = N_eval + 1;
        fprintf(['  Line Search Backtrack Iteration: lambda = %.4e, ' ...
            'F(lambda) = %.4e\n'], lambda, F(lambda));
    end

    % Warning to handle invalid or problematic results
    if isnan(F(lambda)) || F(lambda) > F0
        warning(['Potential issue with the line search: ' ...
            'F(lambda) = %.4e, F(0) = %.4e'], F(lambda), F0);
    end

    return;
end
```

script.m

```
%% Line Search Subroutine Test
a_values = [2, -2, 5, -5, 10, -10]; % Test cases for a
initial_guess = 0.1;                % Initial guess for lambda

% Initialize total function evaluations counter
total_N_eval = 0;

for i = 1:length(a_values)
    a = a_values(i); % Assign the current test case
    fprintf('Test %d, a = %d:\n', i, a); % Print current test info

    F = @(lambda) (1 - 10^a * lambda)^2; % Define the test function

    % Initialize test-specific function evaluation counter
    test_N_eval = 0;

    % Perform line search and track function evaluations
    % Start evaluations from 0 for this test:
    [lambda, test_N_eval] = line_search(F, initial_guess, 0);
    % Add to the total evaluation count:
    total_N_eval = total_N_eval + test_N_eval;

    % Calculate expected lambda
    expected_lambda = 1 / 10^a;
    error = abs(lambda - expected_lambda);
    F_lambda = F(lambda);

    % Print results
    fprintf('Computed lambda = %.5f\n', lambda);
    fprintf('Expected lambda = %.5f\n', expected_lambda);
    fprintf('Error = %.5e\n', error);
    fprintf('F(lambda) = %.5e (should be close to 0)\n', F_lambda);
    fprintf('Function evaluations in this test = %d\n\n', test_N_eval);
end

fprintf('Total function evaluations in all tests = %d\n', total_N_eval);
%% Test 1
[t,y] = data1;
[x,N_eval,N_iter,normg] = gaussnewton(@phi2,t,y,[1;2;3;4],1e-4,1,1);

%% Diverse initial guesss test
[t,y] = data1;
t1 = [0; 0; 0; 0;];         % Neutral start
t2 = [100; 100; 100; 100];  % Far-from-solution guess
t3 = [-10; 0; -10; 0;];     % Negative values
                            % (x2 and x4 must >=0 for phi2, x2 must >= 0 for phi1)

t_choice = t3;              % Current test

[x,N_eval,N_iter,normg] = gaussnewton(@phi2,t,y,t_choice,1e-4,1,1);

%% Task - phi1, data1
[t,y] = data1;
[x,N_eval,N_iter,normg] = gaussnewton(@phi1,t,y,[1;2;3;4],1e-4,1,1);

%% Task - phi1, data2
[t,y] = data2;
[x,N_eval,N_iter,normg] = gaussnewton(@phi1,t,y,[1;2;3;4],1e-4,1,1);

%% Task - phi2, data1 (Test 1)
[t,y] = data1;
[x,N_eval,N_iter,normg] = gaussnewton(@phi2,t,y,[1;2;3;4],1e-4,1,1);
```

```
%% Task - phi2, data2
[t,y] = data2;
[x,N_eval,N_iter,normg] = gaussnewton(@phi2,t,y,[1;2;3;4],1e-4,1,1);

%% Fit phi1 to data1/data2 for multiple starting points
% Format results_phi_data: initial guesses, final values, N_eval, N_iter, normg
[t1, y1] = data1;
[t2, y2] = data2;

% Define range of starting points
num_start_points = 4;                          % Number of different starting points
x1_range = linspace(0.1, 20, num_start_points);
x2_range = linspace(0.01, 10, num_start_points); % Reminder: x2>0 required

% Loop over different starting points for data1
results_phi1_data1 = [];
fprintf('Fitting phi1 to data1:\n');
for x1 = x1_range
    for x2 = x2_range
        x0 = [x1; x2]; % Initial guess
        [x, N_eval, N_iter, normg] = gaussnewton(@phi1, t1, y1, x0, 1e-4, 0, 0);
        results_phi1_data1 = [results_phi1_data1; x0', x', N_eval, N_iter, normg];
        fprintf(['Initial: [%.2f, %.2f], Final: [%.2f, %.2f], ' ...
            'Iterations: %d, Normg: %.2e\n'], ...
            x1, x2, x(1), x(2), N_iter, normg);
    end
end

% Loop over different starting points for data2
results_phi1_data2 = [];
fprintf('\nFitting phi1 to data2:\n');
for x1 = x1_range
    for x2 = x2_range
        x0 = [x1; x2]; % Initial guess
        [x, N_eval, N_iter, normg] = gaussnewton(@phi1, t2, y2, x0, 1e-4, 0, 0);
        results_phi1_data2 = [results_phi1_data2; x0', x', N_eval, N_iter, normg];
        fprintf(['Initial: [%.2f, %.2f], Final: [%.2f, %.2f], ' ...
            'Iterations: %d, Normg: %.2e\n'], ...
            x1, x2, x(1), x(2), N_iter, normg);
    end
end

%% Fit phi2 to data1/data2 for multiple starting points
% Format results_phi_data: initial guesses, final values, N_eval, N_iter, normg
[t1, y1] = data1;
[t2, y2] = data2;

% Define the range of starting points
num_start_points = 4;                          % Number of different starting points
x1_range = linspace(0.1, 20, num_start_points);
x2_range = linspace(0.01, 10, num_start_points); % Reminder x2>0 required
x3_range = linspace(0.1, 20, num_start_points);
x4_range = linspace(0.01, 10, num_start_points); % Reminder: x4>0 required

% Loop over different starting points for data1
results_phi2_data1 = [];
fprintf('\nFitting phi2 to data1:\n');
for x1 = x1_range
    for x2 = x2_range
        for x3 = x3_range
            for x4 = x4_range
                x0 = [x1; x2; x3; x4]; % Initial guess
                [x, N_eval, N_iter, normg] = gaussnewton(@phi2, t1, y1, x0, 1e-4, 0, 0);
                results_phi2_data1 = [results_phi2_data1; ...
                    x0', x', N_eval, N_iter, normg];
                fprintf(['Initial: [%.2f, %.2f, %.2f, %.2f], ' ...
                    'Final: [%.2f, %.2f, %.2f, %.2f], ' ...
                    'Iterations: %d, Normg: %.2e\n'], ...
                    x0(1), x0(2), x0(3), x0(4), x(1), x(2), x(3), x(4), N_iter, normg);
            end
        end
    end
end

% Fit phi2 to data2 using the best results from phi1 on data2
fprintf('\nFitting phi2 to data2:\n');
results_phi2_data2 = [];

% Find the best x1 and x2 from phi1 on data2 (minimum normg)
[min_value, best_index] = min(results_phi1_data2(:, end));
best_x1_x2 = results_phi1_data2(best_index, 3:4); % Extract best [x1, x2]
best_x1 = best_x1_x2(1);
best_x2 = best_x1_x2(2);

% Iterate over x3 and x4 while keeping x1 and x2 fixed
for x3 = x3_range
    for x4 = x4_range
        x0 = [best_x1; best_x2; x3; x4]; % Use best x1 and x2, vary x3 and x4
        [x, N_eval, N_iter, normg] = gaussnewton(@phi2, t2, y2, x0, 1e-4, 0, 0);
        results_phi2_data2 = [results_phi2_data2; x0', x', N_eval, N_iter, normg];
        fprintf(['Initial: [%.2f, %.2f, %.2f, %.2f], ' ...
            'Final: [%.2f, %.2f, %.2f, %.2f], ' ...
            'Iterations: %d, Normg: %.2e\n'], ...
            x0(1), x0(2), x0(3), x0(4), x(1), x(2), x(3), x(4), N_iter, normg);
    end
end
```