Economic Load Dispatch Using Newton Method

Power System Optimization

March 15, 2025

Contents

1 Introduction

This document provides a comprehensive explanation of an Economic Load Dispatch (ELD) solution using the Newton method. ELD is a fundamental optimization problem in power systems that aims to allocate generation among multiple generators to meet the load demand while minimizing the total operating cost. The implementation accounts for power losses in the system and enforces generator constraints.

The solution is divided into three main components:

1. Main ELD code
2. Newton method optimization function
3. Input data file

2 Problem Formulation

2.1 Objective Function

The objective of ELD is to minimize the total generation cost:

$$\min F_T = \sum_{i=1}^{N} F_i(P_{G_i}) \tag{1}$$

where $F_i(P_{G_i}) = a_i P_{G_i}^2 + b_i P_{G_i} + c_i$ is the cost function of generator i.

2.2 Constraints

The optimization is subject to the following constraints:

1. **Power Balance Constraint**:

$$\sum_{i=1}^{N} P_{G_i} = P_D + P_L \tag{2}$$

where P_D is the total load demand and P_L is the total transmission loss.

2. **Generator Limits**:

$$P_{G_i}^{\min} \leq P_{G_i} \leq P_{G_i}^{\max} \tag{3}$$

where $P_{G_i}^{\min}$ and $P_{G_i}^{\max}$ are the minimum and maximum generation limits of generator i.

2.3 Loss Model

The transmission losses are modeled using simplified loss coefficients:

$$P_{L_i} = \alpha_i \cdot P_{G_i}^2 \tag{4}$$

where α_i is the loss coefficient for generator i.

The penalty factor for generator i is calculated as:

$$PF_i = \frac{1}{1 - 2\alpha_i P_{G_i}} \tag{5}$$

3 Data File Implementation

The data file defines the system parameters, including generator cost coefficients, operating limits, and loss coefficients.

```
%% ELD_data.m - Economic Load Dispatch Data
% This file contains the data for the Economic Load Dispatch
  problem
% including generator parameters and system constraints.

% Generator parameters:
% Column 1: a - Quadratic cost coefficient ($/MW^2h)
% Column 2: b - Linear cost coefficient ($/MWh)
% Column 3: c - Constant cost coefficient ($/h)
% Column 4: pg_min - Minimum generation limit (MW)
% Column 5: pg_max - Maximum generation limit (MW)
% Column 6: Initial generation (MW) - not used in the program
% Column 7: ploss_coeff - Loss coefficient for each generator

% Generator data: [a, b, c, pg_min, pg_max, pg_initial,
  ploss_coeff]
PG_data = [
    0.004, 5.3, 500, 200, 450, 0, 0.00003;
    0.006, 5.5, 400, 150, 350, 0, 0.00009;
    0.009, 5.8, 200, 100, 225, 0, 0.00012
];

% Note: The simplified loss formula used is:
% ploss_i = ploss_coeff_i * (pg_i)^2
% Total ploss = sum(ploss_i) for all generators
```

Listing 1: ELD_data.m - Economic Load Dispatch Data

4 Newton Method Function

The Newton method function uses iterative approach to solve the ELD problem by finding the optimal generator outputs and system incremental cost (lambda).

```
function [pg_final_calculated, lambda_final_calculated] =
    newton_method_function(pd, lambda, N, a, b, pg_min, pg_max
    , ploss_coeff, pg, ploss, pf)
    % Initialize variables
    error_tolerance = 0.0001;
    max_iterations = 50;

    % Start with current values
    pg_final_calculated = pg;
    lambda_final_calculated = lambda;

    for iter = 1:max_iterations
        % Calculate updated loss and penalty factors based on
     current generation
        ploss_current = zeros(1, N);
        pf_current = zeros(1, N);
        for j = 1:N
            pf_current(j) = 1/(1-(2*pg_final_calculated(j)*
    ploss_coeff(j)));
            ploss_current(j) = (pg_final_calculated(j)^2)*
    ploss_coeff(j);
        end

        % Build gradient vector (mismatch equations)
        gradient_vector = zeros(N+1, 1);
        for j = 1:N
            gradient_vector(j) = (2*a(j)*pg_final_calculated(
    j)) + b(j) - (lambda_final_calculated/pf_current(j));
        end
        gradient_vector(N+1) = pd + sum(ploss_current) - sum(
    pg_final_calculated);

        % Check convergence on both optimality conditions and
     power balance
        if max(abs(gradient_vector(1:N))) < error_tolerance
    && abs(gradient_vector(N+1)) < error_tolerance
            break;
        end

        % Build Jacobian matrix
        jacobian_matrix = zeros(N+1, N+1);
        for k = 1:N
            % Diagonal elements for generators
```

```
        jacobian_matrix(k, k) = 2*a(k);

        % Lambda column
        jacobian_matrix(k, N+1) = -1/pf_current(k);

        % Power balance row - include the effect of losses
        dloss_dpg = 2*ploss_coeff(k)*pg_final_calculated(k);
        jacobian_matrix(N+1, k) = dloss_dpg - 1;
    end
    jacobian_matrix(N+1, N+1) = 0;

    % Solve for correction vector using \operator (more stable than inv())
    correction_vector = -jacobian_matrix \ gradient_vector;

    % Apply corrections with damping factor to improve convergence
    damping = 1.0;  % Can be reduced if convergence is difficult

    % Update lambda first
    lambda_final_calculated = lambda_final_calculated + damping * correction_vector(N+1);

    % Then update generator outputs with constraints
    for l = 1:N
        % Update with damping
        pg_final_calculated(l) = pg_final_calculated(l) + damping * correction_vector(l);

        % Enforce generator limits
        if pg_final_calculated(l) < pg_min(l)
            pg_final_calculated(l) = pg_min(l);
        elseif pg_final_calculated(l) > pg_max(l)
            pg_final_calculated(l) = pg_max(l);
        end
    end

    % If we've reached max iterations, try to enforce power balance directly
    if iter == max_iterations
        % Calculate current losses
        total_losses = 0;
        for j = 1:N
            total_losses = total_losses + (pg_final_calculated(j)^2)*ploss_coeff(j);
```

```
         end

         % Calculate required generation
         required_generation = pd + total_losses;
         current_generation = sum(pg_final_calculated);

         % Adjust generation if needed and possible
         if abs(required_generation - current_generation) > error_tolerance
             shortage = required_generation - current_generation;

             % Find generators that can be adjusted
             adjustable_gens = [];
             for j = 1:N
                 if shortage > 0 && pg_final_calculated(j) < pg_max(j)
                     adjustable_gens = [adjustable_gens j];
                 elseif shortage < 0 && pg_final_calculated(j) > pg_min(j)
                     adjustable_gens = [adjustable_gens j];
                 end
             end

             % Distribute the shortage among adjustable generators
             if ~isempty(adjustable_gens)
                 adjustment = shortage / length(adjustable_gens);
                 for j = adjustable_gens
                     pg_final_calculated(j) = pg_final_calculated(j) + adjustment;

                     % Re-check limits after adjustment
                     if pg_final_calculated(j) < pg_min(j)
                         pg_final_calculated(j) = pg_min(j);
                     elseif pg_final_calculated(j) > pg_max(j)
                         pg_final_calculated(j) = pg_max(j);
                     end
                 end
             end
         end
     end
 end
```

```
end
```

Listing 2: newton_method_function.m - Newton Method Implementation

4.1 Function Description

The Newton method function takes the following inputs:

- `pd`: Power demand
- `lambda`: Initial lambda (incremental cost) value
- `N`: Number of generators
- `a`, `b`: Cost function coefficients
- `pg_min`, `pg_max`: Generator limits
- `ploss_coeff`: Loss coefficients
- `pg`: Initial generator outputs
- `ploss`: Initial power losses
- `pf`: Initial penalty factors

It returns:

- `pg_final_calculated`: Optimized generator outputs
- `lambda_final_calculated`: Final lambda value (system incremental cost)

4.2 Algorithm Steps

1. **Initialization**: Set up variables and starting values.
2. **Iterative Process**:
   - Calculate updated loss and penalty factors
   - Build the gradient vector (optimality conditions and power balance)
   - Check for convergence
   - Build the Jacobian matrix

- Solve for correction vector
- Update lambda and generator outputs with constraints

3. **Final Adjustment**: If needed, enforce power balance directly by adjusting available generators.

4.3 Key Components

4.3.1 Gradient Vector

The gradient vector contains the mismatch equations:

- For generators (1 to N): $\frac{\partial F_i}{\partial P_{G_i}} - \frac{\lambda}{PF_i} = 0$
- For power balance (N+1): $P_D + P_L - \sum P_{G_i} = 0$

4.3.2 Jacobian Matrix

The Jacobian matrix contains the derivatives of the gradient vector:

- Diagonal elements (k,k): $\frac{\partial^2 F_k}{\partial P_{G_k}^2} = 2a_k$
- Lambda column (k,N+1): $\frac{\partial}{\partial \lambda}(\frac{\lambda}{PF_k}) = -\frac{1}{PF_k}$
- Power balance row (N+1,k): $\frac{\partial P_L}{\partial P_{G_k}} - 1 = 2\alpha_k P_{G_k} - 1$

4.3.3 Correction Vector

The correction vector is calculated by solving the linear system:

$$J \cdot \Delta x = -g \tag{6}$$

where J is the Jacobian matrix, g is the gradient vector, and Δx contains the corrections for generator outputs and lambda.

5 Main ELD Program

The main program coordinates the overall solution process, handles iterations, and reports results.

```
clear all
clc
ELD_Data  % Load the data

% Extract generator parameters
N = size(PG_data, 1);
a = PG_data(:, 1);
b = PG_data(:, 2);
c = PG_data(:, 3);
pg_min = PG_data(:, 4);
pg_max = PG_data(:, 5);
ploss_coeff = PG_data(:, 7);
pd = 975;  % Load demand

% Initialize variables with a feasible starting point
pg = zeros(1, N);
total_min = sum(pg_min);
total_max = sum(pg_max);

if pd < total_min
    error('Demand is less than minimum generation capacity');
elseif pd > total_max
    error('Demand exceeds maximum generation capacity');
else
    % Distribute load proportionally between min and max limits
    for i = 1:N
        pg(i) = pg_min(i) + (pg_max(i) - pg_min(i)) * (pd - total_min) / (total_max - total_min);
    end
end

% Better initial lambda estimate based on average marginal cost
lambda_init = 0;
for i = 1:N
    lambda_init = lambda_init + 2*a(i)*pg(i) + b(i);
end
lambda = lambda_init / N;

error_tolerance = 0.01;  % Tolerance for convergence
max_iterations = 100;  % Maximum iterations

% Initialize loss and penalty factors
ploss = zeros(1, N);
pf = zeros(1, N);

% Calculate initial ploss and pf
for i = 1:N
```

```
    pf(i) = 1/(1-(2*pg(i)*ploss_coeff(i)));
    ploss(i) = (pg(i)^2)*ploss_coeff(i);
end

total_ploss = sum(ploss);
fprintf('Initial: Gen: %.2f MW, Demand: %.2f MW, Loss: %.2f MW, Balance: %.2f MW\n', sum(pg), pd, total_ploss, sum(pg) - (pd + total_ploss));

% Main iteration loop
for iter = 1:max_iterations
    % Call Newton method to update pg and lambda
    [pg_new, lambda_new] = newton_method_function(pd, lambda, N, a, b, pg_min, pg_max, ploss_coeff, pg, ploss, pf);

    % Calculate new loss and penalty factors
    ploss_new = zeros(1, N);
    pf_new = zeros(1, N);
    for j = 1:N
        pf_new(j) = 1/(1-(2*pg_new(j)*ploss_coeff(j)));
        ploss_new(j) = (pg_new(j)^2)*ploss_coeff(j);
    end

    % Calculate changes for convergence check
    total_ploss_new = sum(ploss_new);
    power_balance = sum(pg_new) - (pd + total_ploss_new);

    % Print iteration details
    fprintf('Iter %2d: Gen: %.2f MW, Loss: %.2f MW, Balance: %.6f MW, Lambda: %.6f\n', iter, sum(pg_new), total_ploss_new, power_balance, lambda_new);

    % Update values for next iteration
    pg = pg_new;
    lambda = lambda_new;
    ploss = ploss_new;
    pf = pf_new;

    % Check convergence on power balance
    if abs(power_balance) < error_tolerance
        fprintf('\nConverged after %d iterations!\n', iter);
        break;
    end

    % If we're at the last iteration and still not converged, try one final adjustment
    if iter == max_iterations
        fprintf('\nReached maximum iterations. Performing final adjustment...\n');
```

```matlab

        % Calculate current total loss
        total_ploss = sum(ploss);

        % Calculate required total generation
        required_generation = pd + total_ploss;

        % Find generators not at their limits
        adjustable_gens = [];
        for i = 1:N
            if pg(i) > pg_min(i) && pg(i) < pg_max(i)
                adjustable_gens = [adjustable_gens i];
            end
        end

        if ~isempty(adjustable_gens)
            % Calculate current imbalance
            current_imbalance = sum(pg) - required_generation;

            % Distribute the adjustment among available generators
            adjustment_per_gen = current_imbalance / length(adjustable_gens);

            for i = adjustable_gens
                pg(i) = pg(i) - adjustment_per_gen;

                % Ensure limits are respected
                if pg(i) < pg_min(i)
                    pg(i) = pg_min(i);
                elseif pg(i) > pg_max(i)
                    pg(i) = pg_max(i);
                end

                % Update loss for this generator
                ploss(i) = (pg(i)^2) * ploss_coeff(i);
            end

            % Recalculate power balance
            total_ploss = sum(ploss);
            power_balance = sum(pg) - (pd + total_ploss);

            fprintf('After adjustment: Gen: %.2f MW, Loss: %.2f MW, Balance: %.6f MW\n', sum(pg), total_ploss, power_balance);
        end
    end
end
```

```

% Print final results
fprintf('\nFinal Results:\n');
fprintf('Generator\tOutput (MW)\tMin (MW)\tMax (MW)\tMarginal
    Cost ($/MWh)\n');
for i = 1:N
    fprintf('%d\t\t%.2f\t\t%.2f\t\t%.2f\t\t%.6f\n', i, pg(i),
    pg_min(i), pg_max(i), 2*a(i)*pg(i) + b(i));
end

fprintf('\nTotal generation: %.2f MW\n', sum(pg));
fprintf('Total demand: %.2f MW\n', pd);
fprintf('Total losses: %.2f MW\n', sum(ploss));
fprintf('Power balance: %.6f MW\n', sum(pg) - (pd + sum(ploss
  )));
fprintf('Final lambda (system marginal cost): %.6f $/MWh\n',
  lambda);

% Calculate total cost
total_cost = 0;
for i = 1:N
    gen_cost = a(i)*pg(i)^2 + b(i)*pg(i) + c(i);
    total_cost = total_cost + gen_cost;
    fprintf('Generator %d cost: $%.2f/h\n', i, gen_cost);
end
fprintf('Total system cost: $%.2f/h\n', total_cost);
```

Listing 3: main_ELD.m - Main Economic Load Dispatch Program

5.1 Main Program Structure

The main program follows these general steps:

1. **Data Preparation and Initialization**:
    - Load generator data
    - Extract parameters (cost coefficients, limits, etc.)
    - Initialize generator outputs and lambda
    - Set up convergence criteria
2. **Initial Calculations**:
    - Calculate initial penalty factors and losses
    - Verify feasibility of the problem
3. **Main Iteration Loop**:

- Call Newton method function to update generator outputs and lambda
- Calculate new losses and penalty factors
- Check for convergence based on power balance
- Update all variables for next iteration

4. **Final Adjustment** (if needed):
   - Find adjustable generators (not at their limits)
   - Distribute any remaining imbalance
   - Recalculate power balance

5. **Results Reporting**:
   - Print detailed information about generator outputs
   - Report system performance metrics (total generation, losses, etc.)
   - Calculate and display total cost

5.2 Key Components

5.2.1 Initialization Strategy

The program uses a smart initialization strategy:

- Distributes initial generation proportionally between minimum and maximum limits
- Estimates initial lambda based on average marginal costs

5.2.2 Convergence Criteria

The primary convergence criterion is the power balance:

$$|P_G - (P_D + P_L)| < \epsilon \tag{7}$$

where ϵ is the error tolerance (0.01 MW in this implementation).

5.2.3 Variable Updates

Between iterations, all key variables are properly updated:

- Generator outputs (pg)
- System incremental cost (lambda)
- Power losses (ploss)
- Penalty factors (pf)

5.2.4 Final Adjustment Mechanism

If normal iterations don't achieve power balance, a direct adjustment mechanism:

1. Identifies generators not at their limits
2. Calculates the remaining imbalance
3. Distributes the adjustment among available generators
4. Enforces generator limits
5. Recalculates the final power balance

6 Theoretical Background

6.1 Newton Method for ELD

The Newton method for ELD is based on solving the Lagrangian function:

$$L = \sum_{i=1}^{N} F_i(P_{G_i}) - \lambda \left(\sum_{i=1}^{N} P_{G_i} - P_D - P_L \right) \tag{8}$$

The first-order optimality conditions are:

$$\frac{\partial L}{\partial P_{G_i}} = \frac{\partial F_i}{\partial P_{G_i}} - \lambda \cdot \frac{\partial}{\partial P_{G_i}} \left(1 - \frac{\partial P_L}{\partial P_{G_i}} \right) = 0 \tag{9}$$

When including transmission losses, this becomes:

$$\frac{\partial F_i}{\partial P_{G_i}} = \lambda \cdot PF_i \tag{10}$$

where PF_i is the penalty factor for generator i.

6.2 Handling Generator Limits

When a generator output reaches its limit, that generator is fixed at the limit, and the optimization continues with the remaining generators. This is implemented by enforcing the limits after each update:

$$P_{G_i} = \begin{cases} P_{G_i}^{\min} & \text{if } P_{G_i} < P_{G_i}^{\min} \\ P_{G_i} & \text{if } P_{G_i}^{\min} \leq P_{G_i} \leq P_{G_i}^{\max} \\ P_{G_i}^{\max} & \text{if } P_{G_i} > P_{G_i}^{\max} \end{cases} \tag{11}$$

7 Optimization Results and Analysis

7.1 Convergence Behavior

The implementation typically converges within a few iterations for well-behaved systems. Convergence is monitored by:

1. Tracking the power balance error
2. Monitoring changes in power losses
3. Verifying that optimality conditions are satisfied

7.2 Analysis of Results

The final results provide comprehensive information:

- Individual generator outputs and their marginal costs
- Total system generation and demand
- Total transmission losses
- Final power balance verification
- System marginal cost (lambda)
- Total generation cost

7.3 Optimal Generation Dispatch

The optimal dispatch follows the equal incremental cost principle, adjusted by penalty factors:

$$\frac{1}{PF_1}\frac{\partial F_1}{\partial P_{G_1}} = \frac{1}{PF_2}\frac{\partial F_2}{\partial P_{G_2}} = \cdots = \frac{1}{PF_N}\frac{\partial F_N}{\partial P_{G_N}} = \lambda \tag{12}$$

Generators with lower costs and lower loss impacts are dispatched more heavily, subject to their operating limits.

8 Conclusion

The Economic Load Dispatch solution using the Newton method provides an efficient approach to optimally allocate generation among multiple generators while accounting for transmission losses and enforcing generator constraints. The implementation presented here demonstrates a robust solution that achieves:

1. **Cost Optimization**: Minimizes the total generation cost
2. **Constraint Satisfaction**: Respects generator operating limits
3. **Power Balance**: Ensures generation matches demand plus losses
4. **Numerical Stability**: Uses techniques to enhance convergence

This implementation can be used as a foundation for more complex power system optimization problems and can be extended to include additional constraints and considerations specific to power system operations.

A MATLAB Files Summary

To implement the Economic Load Dispatch solution presented in this document, three MATLAB files are needed:

1. `ELD_Data.m`: Contains generator parameters, cost coefficients, and loss coefficients.
2. `newton_method_function.m`: Implements the Newton method for ELD optimization.
3. `main_ELD.m`: Main program that coordinates the overall solution process.

These files should be placed in the same directory and executed by running the `main_ELD.m` script in MATLAB.