Economic Load Dispatch Using Reduced Gradient Method

March 15, 2025

Contents

1 Introduction

Economic Load Dispatch (ELD) is a fundamental optimization problem in power system operation that aims to determine the optimal output of multiple generating units to meet a specific load demand at the lowest possible cost while satisfying various operational constraints. This document explains the implementation of the Reduced Gradient Method for solving the ELD problem, taking into account transmission line losses.

2 Problem Formulation

2.1 Objective Function

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

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

where:

- N is the number of generators
- P_{G_i} is the power output of generator i
- $C_i(P_{G_i})$ is the cost function of generator i

The cost function for each generator is typically modeled as a quadratic function:

$$C_i(P_{G_i}) = a_i P_{G_i}^2 + b_i P_{G_i} + c_i \tag{2}$$

where a_i, b_i, and c_i are the cost coefficients.

2.2 Constraints

2.2.1 Power Balance Constraint

The total power generated must equal the sum of the total demand and the transmission line losses:

$$\sum_{i=1}^{N} P_{G_i} = P_D + P_L \tag{3}$$

where:

- P_D is the total load demand
- P_L is the total transmission line losses

2.2.2 Generator Capacity Constraints

The power output of each generator is bounded by its minimum and maximum limits:

$$P_{G_i}^{\min} \leq P_{G_i} \leq P_{G_i}^{\max} \tag{4}$$

2.3 Transmission Line Losses

Transmission line losses are modeled using simplified B-coefficients:

$$P_L = \sum_{i=1}^{N} B_{ii} P_{G_i}^2 \tag{5}$$

where B_{ii} are the loss coefficients for each generator.

3 The Reduced Gradient Method

3.1 Mathematical Formulation

The reduced gradient method is an optimization technique for solving constrained optimization problems. For the ELD problem, we can form the Lagrangian:

$$L(P_G, \lambda) = \sum_{i=1}^{N} C_i(P_{G_i}) + \lambda \left(\sum_{i=1}^{N} P_{G_i} - P_D - P_L \right) \tag{6}$$

where λ is the Lagrange multiplier.

The optimality conditions require:

$$\frac{\partial L}{\partial P_{G_i}} = \frac{dC_i}{dP_{G_i}} + \lambda \left(1 - \frac{\partial P_L}{\partial P_{G_i}} \right) = 0, \quad i = 1, 2, \ldots, N \tag{7}$$

This leads to the concept of the penalty factor for each generator:

$$PF_i = \frac{1}{1 - \frac{\partial P_L}{\partial P_{G_i}}} \tag{8}$$

With the simplified loss model, the penalty factor becomes:

$$PF_i = \frac{1}{1 - 2B_{ii}P_{G_i}} \quad (9)$$

The reduced gradient method classifies variables into dependent and independent sets. In the ELD problem, we typically select one generator as the dependent variable (usually the last one) and adjust it to satisfy the power balance constraint.

3.2 Algorithm Steps

1. Initialize generator outputs P_G and Lagrange multiplier λ
2. Calculate initial losses P_L
3. Compute penalty factors PF
4. For each iteration:
   (a) Select a dependent generator (typically the last one)
   (b) Adjust its output to satisfy the power balance constraint
   (c) Calculate the reduced gradient for each independent generator
   (d) Update generator outputs using gradient descent
   (e) Check for generator limit violations and adjust accordingly
   (f) Recalculate losses and update the dependent generator
   (g) Check for convergence

4 Code Implementation

4.1 Main Program Structure

The main program is structured as follows:

1. Data initialization
2. Finding a feasible initial solution
3. Main iteration loop calling the reduced gradient function
4. Final adjustments to ensure power balance
5. Results display and visualization

4.1.1 Data Initialization

```
% Load ELD data
% Format: [a, b, c, pg_min, pg_max, pgi_guess,
    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];

N = length(PG_data(:,1)); % Number of generators
a = PG_data(:,1);  % Quadratic cost coefficient
b = PG_data(:,2);  % Linear cost coefficient
c = PG_data(:,3);  % Constant cost coefficient
pg_min = PG_data(:,4);  % Minimum generation limit
pg_max = PG_data(:,5);  % Maximum generation limit
ploss_coeff = PG_data(:,7);  % Loss coefficients

pd = 975;  % Demand value in MW
```

Listing 1: Data initialization for the ELD problem

4.1.2 Finding Feasible Initial Solution

The code initializes the generators with a feasible solution by:

1. Setting most generators to their maximum capacity
2. Estimating initial losses
3. Adjusting the swing generator to balance the system
4. Checking if the swing generator violates its limits and redistributing if necessary

```
% Initialize with generators at maximum except the last
    one
pg = zeros(N, 1);
for i = 1:N-1
    pg(i) = pg_max(i);
end

% Calculate initial losses estimate
initial_loss_estimate = pd * 0.03; % Assume 3% losses
    initially
```

```
target_gen = pd + initial_loss_estimate;

% Adjust to meet the target generation + estimated losses
if sum(pg(1:N-1)) > target_gen
    % Sort by marginal cost (descending) to reduce most expensive first
    [~, cost_order] = sort([2*a(1:N-1).*pg_max(1:N-1) + b(1:N-1)], 'descend');

    excess = sum(pg(1:N-1)) - target_gen;
    for idx = 1:N-1
        i = cost_order(idx);
        reduction = min(excess, pg(i) - pg_min(i));
        pg(i) = pg(i) - reduction;
        excess = excess - reduction;
        if excess <= 0
            break;
        end
    end
end

% Set the swing generator to balance
pg(N) = pd - sum(pg(1:N-1)); % Initial estimate without losses

% Calculate initial losses and update swing generator
ploss = sum(ploss_coeff .* pg.^2);
pg(N) = pd + ploss - sum(pg(1:N-1)); % Update with losses
```

Listing 2: Finding a feasible initial solution

4.1.3 Main Iteration Loop

The main iteration loop calls the reduced gradient function and updates parameters until convergence:

```
for iteration = 1:max_iterations
    % Call reduced gradient function
    [pg, lambda, ploss_new] = reduced_gradient_function(alpha, N, error_tolerance_reduced_gradient, ...
```

```
                                                    a, b, c,
                                                       lambda,
                                                       ploss_coeff,
                                                       pd, ploss,
                                                       pf,
                                                       pg_old,
                                                       pg_min,
                                                       pg_max);

    % Update penalty factors
    pf_new = 1./(1 - 2*pg.*ploss_coeff);

    % Calculate difference in losses
    diff_ploss = sum(ploss_new) - sum(ploss);

    % Check convergence criteria
    is_converged_loss = (abs(diff_ploss) <
       error_tolerance_ploss_diff);
    is_within_limits = ~limits_violated;
    is_balanced = (abs(power_balance) < 0.1);

    if is_converged_loss && is_within_limits &&
       is_balanced
        break;
    end

    % Update for next iteration
    ploss = ploss_new;
    pf = pf_new;
    pg_old = pg;

    % Adaptive step size adjustment
    if iteration > 10
        if abs(diff_ploss) >
           error_tolerance_ploss_diff*10 ||
           abs(power_balance) > 1
            alpha = alpha * 0.95; % Reduce step size
        elseif iteration > 30 && abs(diff_ploss) <
           error_tolerance_ploss_diff*100 &&
           abs(power_balance) < 10
            alpha = alpha * 1.05; % Increase step size
            alpha = min(alpha, 0.01); % Cap step size
        end
```

```
    end
end
```

Listing 3: Main iteration loop

4.2 Reduced Gradient Function

The reduced gradient function implements the core optimization algorithm:

```
function [pg, lambda, ploss_updated] = reduced_gradient_function(alpha, N, error_tolerance, ...
                                                            a, b, c, lambda, ploss_coeff, pd, ploss, pf, pg_old, pg_min, pg_max)
    % Initialize variables
    pg = pg_old;
    gradient_vector = zeros(N+1, 1);

    % Calculate initial losses
    ploss_updated = sum(ploss_coeff .* pg.^2);

    % Reduced gradient method iterations
    max_inner_iterations = 100;
    for iteration = 1:max_inner_iterations
        % Apply generator limits
        for i = 1:N
            if pg(i) < pg_min(i)
                pg(i) = pg_min(i);
            elseif pg(i) > pg_max(i)
                pg(i) = pg_max(i);
            end
        end

        % Recalculate losses
        ploss_updated = sum(ploss_coeff .* pg.^2);

        % Select dependent generator
```

```
        dependent_gen = N;

        % Set dependent generator to balance power
        pg(dependent_gen) = pd + ploss_updated - sum(pg(1:N)) + pg(dependent_gen);

        % Check dependent generator limits and redistribute if necessary
        if pg(dependent_gen) < pg_min(dependent_gen)
            % Handle case where dependent generator is below minimum
            deficit = pg_min(dependent_gen) - pg(dependent_gen);
            pg(dependent_gen) = pg_min(dependent_gen);

            % Find generators that can increase output
            % ... (redistribution logic)

        elseif pg(dependent_gen) > pg_max(dependent_gen)
            % Handle case where dependent generator is above maximum
            excess = pg(dependent_gen) - pg_max(dependent_gen);
            pg(dependent_gen) = pg_max(dependent_gen);

            % Find generators that can decrease output
            % ... (redistribution logic)
        end

        % Recalculate losses
        ploss_updated = sum(ploss_coeff .* pg.^2);

        % Calculate gradients for each generator
        for i = 1:N
            if i == dependent_gen
                gradient_vector(i) = 0;  % Skip dependent generator
                continue;
            end

            % Skip generators at their limits
            if pg(i) <= pg_min(i) && gradient_vector(i) > 0
```

```
            gradient_vector(i) = 0;
            continue;
        elseif pg(i) >= pg_max(i) && gradient_vector(i) < 0
            gradient_vector(i) = 0;
            continue;
        end

        % Calculate marginal costs
        dCost_i = 2*a(i)*pg(i) + b(i);  % Incremental cost of generator i
        dCost_dep = 2*a(dependent_gen)*pg(dependent_gen) + b(dependent_gen);  % Incremental cost of dependent generator

        % Calculate loss sensitivities
        dLoss_i = 2*ploss_coeff(i)*pg(i);  % Change in losses due to generator i
        dLoss_dep = 2*ploss_coeff(dependent_gen)*pg(dependent_gen);  % Change in losses due to dependent generator

        % Calculate penalty factors
        pf_i = 1/(1 - dLoss_i);
        pf_dep = 1/(1 - dLoss_dep);

        % Calculate reduced gradient
        gradient_vector(i) = pf_i * dCost_i - pf_dep * dCost_dep;
    end

    % Power balance constraint gradient
    gradient_vector(N+1) = sum(pg) - (pd + ploss_updated);

    % Update generators using gradient descent
    max_gradient = 0;
    for i = 1:N
        if i == dependent_gen
            continue;  % Skip dependent generator
        end
```

```
            % Only update if not at limits% Only update if not at limits or if gradient pushes away from limit
            if (pg(i) > pg_min(i) && pg(i) < pg_max(i)) || ...
               (pg(i) <= pg_min(i) && gradient_vector(i) < 0) || ...
               (pg(i) >= pg_max(i) && gradient_vector(i) > 0)

                step = alpha * gradient_vector(i);
                pg(i) = pg(i) - step;

                % Apply limits after update
                if pg(i) < pg_min(i)
                    pg(i) = pg_min(i);
                elseif pg(i) > pg_max(i)
                    pg(i) = pg_max(i);
                end
            end

            % Track maximum gradient for convergence check
            max_gradient = max(max_gradient, abs(gradient_vector(i)));
        end

        % Update lambda (Lagrange multiplier)
        lambda = lambda + alpha * gradient_vector(N+1);

        % Recalculate dependent generator and losses
        ploss_updated = sum(ploss_coeff .* pg.^2);
        pg(dependent_gen) = pd + ploss_updated - sum(pg(1:N)) + pg(dependent_gen);

        % Apply limits to dependent generator
        if pg(dependent_gen) < pg_min(dependent_gen)
            pg(dependent_gen) = pg_min(dependent_gen);
        elseif pg(dependent_gen) > pg_max(dependent_gen)
            pg(dependent_gen) = pg_max(dependent_gen);
        end

```

```
        % Check convergence
        power_balance = sum(pg) - (pd + ploss_updated);
        if max_gradient < error_tolerance &&
           abs(power_balance) < error_tolerance
            break;
        end
    end

    % Final recalculation of losses
    ploss_updated = sum(ploss_coeff .* pg.^2);
end
```

Listing 4: Reduced gradient function implementation

5 Key Algorithmic Features

5.1 Initial Solution Feasibility

The algorithm starts with a feasible solution by:

- Setting generators to their maximum capacity (except the last one)
- Estimating losses
- Balancing the system using the swing generator
- Checking and adjusting if the swing generator violates its limits

This initialization approach ensures that the algorithm begins from a valid point in the feasible region, which helps with convergence. Starting with generators at their maximum capacities allows the algorithm to determine if capacity constraints are binding and provides a known reference point for adjustments.

5.2 Adaptive Step Size

The algorithm uses an adaptive step size to improve convergence:

- Decreases step size when changes in losses are large or power balance is poor
- Increases step size when convergence is slow but stable
- Caps the maximum step size to prevent oscillation

The adaptive step size mechanism is crucial for balancing between speed and stability of convergence. When the solution is far from optimal, larger steps help reach the vicinity of the optimum quickly. As the solution approaches the optimum, smaller steps provide more precise convergence without overshooting.

5.3 Penalty Factors

Penalty factors account for the effect of losses on incremental costs:

$$PF_i = \frac{1}{1 - 2B_{ii}P_{G_i}} \tag{10}$$

These factors adjust the incremental costs to reflect the true cost of delivering power to the load. Penalty factors are essential in systems with significant transmission losses, as they ensure that the optimization accounts for the additional generation needed to compensate for these losses. Without penalty factors, the solution would underestimate the true cost of generation.

5.4 Dependent Generator Selection

The algorithm selects one generator (usually the last one) as the dependent variable, which is adjusted to satisfy the power balance constraint. If this generator violates its limits, the excess or deficit is redistributed among other generators based on their incremental costs.

Choosing a dependent generator reduces the dimensionality of the problem and ensures that the power balance constraint is always satisfied during the optimization process. The redistribution mechanism handles cases where the dependent generator cannot alone satisfy the balance constraint due to its capacity limits.

5.5 Convergence Criteria

The algorithm checks for convergence using multiple criteria:

- Small gradients (optimality)
- Small changes in losses (stability)
- Power balance (feasibility)
- Generators within limits (constraints)

Using multiple convergence criteria ensures that the final solution is not only optimal but also feasible and stable. The algorithm terminates only when all criteria are satisfied, providing a robust solution to the economic load dispatch problem.

6 Example Test Case

The test case provided in the code has the following parameters:

6.1 Generator Data

Generator	a	b	c	Min (MW)	Max (MW)	B
1	0.004	5.3	500	200	450	0.00003
2	0.006	5.5	400	150	350	0.00009
3	0.009	5.8	200	100	225	0.00012

Table 1: Generator Parameters for the Test Case

6.2 System Demand

Total Load Demand: 975 MW

6.3 Expected Results Analysis

For this test case, we would expect the following characteristics in the optimal solution:

- Generator 1 will likely operate at a higher output due to its lower quadratic cost coefficient ($a_1 = 0.004$)
- Generator 3 will likely operate at a lower output due to its higher quadratic cost coefficient ($a_3 = 0.009$)
- The penalty factors will be higher for generators with larger loss coefficients
- The adjusted incremental costs (including penalty factors) should be approximately equal at the optimal operating point
- The total system losses will typically be around 2-5% of the total demand
- The optimal solution should satisfy all generator limits

7 Visualization and Analysis

The code includes visualization of:

- Generator outputs compared to their limits
- Cost curves for each generator
- Incremental cost curves
- Operating points on the cost curves

These visualizations help in understanding the economic operation of the power system and verifying that the solution satisfies the equal incremental cost criterion, adjusted for losses. Visual analysis provides intuitive confirmation that the numerical solution is correct and allows quick identification of potential issues, such as generators operating at their limits.

7.1 Interpretation of Results

When interpreting the results, several key indicators should be examined:

- **Generator Operating Points**: Check if any generators are at their limits. If so, the equal incremental cost criterion may not apply to these generators.
- **Incremental Costs**: The adjusted incremental costs (including penalty factors) should be approximately equal for all generators not operating at their limits. This confirms the optimality of the solution.
- **System Losses**: The percentage of losses relative to the total demand provides an indication of the efficiency of the dispatch. High losses might suggest that a different dispatch strategy could be more economical.
- **Total Generation Cost**: This is the primary objective function and should be minimized. Comparing this cost with alternative dispatch strategies confirms the effectiveness of the optimization.

8 Conclusion

The reduced gradient method provides an effective approach to solving the Economic Load Dispatch problem with transmission line losses. The implementation includes:

1. Proper handling of generator limits
2. Accurate modeling of transmission losses
3. Adaptive step size for improved convergence
4. Multiple convergence criteria for solution quality
5. Visualization tools for result analysis

This implementation can be extended to include additional constraints such as ramp rate limits, prohibited operating zones, and multiple fuels by modifying the gradient calculation and constraint handling.

8.1 Future Enhancements

Several enhancements could further improve the algorithm:

- **Full B-matrix representation**: Incorporating the full B-matrix for loss modeling would provide more accurate results for complex power systems with significant cross-coupling between generators.
- **Valve-point effects**: Including valve-point loading effects in the cost function would provide a more realistic representation of thermal generator characteristics.
- **Multi-objective optimization**: Extending the algorithm to consider both cost and emissions could provide environmentally friendly dispatch solutions.
- **Integration with renewable sources**: Incorporating the stochastic nature of renewable energy sources would make the algorithm applicable to modern power systems with high renewable penetration.
- **Security constraints**: Adding line flow constraints would ensure that the dispatch solution does not violate transmission system security limits.

The reduced gradient method, with its ability to handle constraints effectively, provides a strong foundation for these extensions, making it a valuable tool for power system operation and planning.