SSA 4

Numerical Optimization for Minimal Buckling in Truss Structure

Csaba Benedek

September 22, 2025

Goal

- Use numerical optimization methods to create the most buckling-resistant variation of the latest crane design.

Conclusion

- An iterative multi-point optimization algorithm was used to minimize the buckling distribution factor ($\Gamma = \gamma + 2s_\mu$).
- The latest crane design was processed using the algorithm, with the targeted points located in the upper section where the arm connects to the base.
- The simulation successfully minimized the buckling distribution factor for the design.

Problems

- The optimization algorithm utilizes an 'objective' function that only seeks to minimize the buckling distribution factor.
- This resulted in increased overall loads in the truss structure.

Follow up Steps

- Create a multi-index objective function that incorporates other design requirements with corresponding tunable weights placed upon the importance of each.

Work Division

- The work was divided into three stages:
  1. Create a custom FEM algorithm for use in Python.
  2. Develop a variation and iteration algorithm to optimize the design.
  3. Apply the algorithms to existing designs.
- All components of the work were done by one person, which resulted in difficulties in achieving ideal results.

Time Division

- Developing a custom FEM algorithm in Python. **2 hours**
- Creating a single-point variation algorithm. **2 hours**
- Developing an iterative multi-point optimizer. **2 hour**
- Writing parts of the SSA itself. **2 hour**
- Application to the latest crane design (Crane Design 3) **1 hour**
- Finishing writing and revision **1 hour**

Total of 10 hours

Goal 1 - Development of Numerical Optimization Method

Synopsis of previous work

In SSA 3, expressions were obtained to apply findings about buckling to any crane design at a single element and global level of analysis. Firstly, the utilization ratio, which expresses how close the compressive loading is to the critical buckling load, was defined as:

$$\mu_i = \left| \frac{T_i}{P_{c,i}} \right| , \tag{1}$$

where T_i is the compressive loading and $P_{c,i}$ is the critical buckling load of any element i.

To extend this to a truss structure at a global level and assess the distribution of buckling risk across the entire structure, expressions were created for the weighted buckling index (which is a weighted mean of the utilization ratios) γ, the weighted variance s_μ^2, the coefficient of variation v_μ, and the buckling distribution factor (previously referred to as the safety margin factor), which must satisfy the following:

$$\gamma + 2s_\mu \leq 0.6 \tag{2}$$

These indicators are useful in analyzing the buckling resistance of truss structures and may be derived from only the individual lengths and forces in truss members. Therefore, they serve as the ideal foundation for a numerical optimization approach, where the goal is to minimize the value of the buckling distribution factor.

Additionally, the buckling distribution factor will be expressed using the variable Γ throughout the rest of this report for brevity; thus, it may be expressed as:

$$\Gamma = \gamma + 2s_\mu \tag{3}$$

Numerical optimization methodology

Since the previous buckling indices are calculated using the element lengths and the internal forces within them, any modification to the position of a connection point (node) necessitates a new simulation to determine the updated forces. To determine the lengths of elements, simple trigonometry can be applied to the position of each node. However, obtaining the forces in elements requires lengthy calculations or a simulation.

Although Marc Mentat has previously been used as a basis for obtaining reliable theoretical forces within the past designs, there is no simple or quick way to automatically re-simulate a structure when varying the position of a point. Thus, this numerical approach utilizes a custom Python-based truss solver that may be called as a function when varying the position of points. The code for the entire project has been linked in Appendix A; however, snippets for important details will be included throughout.

Truss solver algorithm

As the algorithm for solving the internal forces in a truss structure is a relatively common problem that has been solved before, an algorithm has been adapted to fit within the context of this project from online sources.[1]

The theoretical foundation for this truss solver has been adapted from a typical stiffness matrix model.[2] The basis comes from Euler-Bernoulli beam theory, where each node in an element has a set of degrees of freedom (DOFs). For 2D, this would typically be 3 DOFs (translation in x, translation in y, and rotation)–however, for the sake of simplicity, only the translational DOFs will be used. The relationship between these DOFs and an element can be used to produce an

[1] Alejo, "GitHub - alejo1630/truss_solver: This Python code solves any 2D truss structure using the joints method," GitHub. https://github.com/alejo1630/truss_solver

[2] S. Moaveni, Finite element analysis: Theory and Application with ANSYS. Prentice Hall, 2008.

element stiffness matrix, which in this case has dimensions of $\mathbf{4 \times 4}$ (2 degrees of freedom for each of 2 nodes in one element and their corresponding relationships). This may be annotated in the following way:

$$\mathbf{k} = k_{local} \begin{bmatrix} c_x^2 & c_x c_y & -c_x^2 & -c_x c_y \\ c_x c_y & c_y^2 & -c_x c_y & -c_y^2 \\ -c_x^2 & -c_x c_y & c_x^2 & c_x c_y \\ -c_x c_y & -c_y^2 & c_x c_y & c_y^2 \end{bmatrix}$$

where c_x and c_y are the direction cosines defined as in terms of the coordinates of nodes 1 and 2 in an element of length L:

$$c_x = \frac{x_2 - x_1}{L} \ , \ c_y = \frac{y_2 - y_1}{L}$$

Additionally, the term k_{local} is factored out from the expression and relates to the material properties. It is given by the Young's modulus E, the cross-sectional area A, and the length L of the element in the following equation:

$$k_{local} = \frac{EA}{L}$$

Next, these individual element stiffness matrices are combined into one global stiffness matrix for the entire truss structure as follows:

$$[K] = \sum_i [k_i]$$

Similarly, the applied forces (or load boundary conditions) can be summed as vectors (with dimension $\mathbf{4 \times 1}$ in this case) as such:

$$\{F\} \sum_i \{f_i\}$$

Once these sums have been computed, the nodal displacements may be solved via the following equation:[3]

$$\{F\} + \{R\} = [K]\{U\} \ ,$$

where $\{F\}$ is the applied force vector, $\{R\}$ is the external reaction force vector, $[K]$ is the global stiffness matrix, and $\{U\}$ is the nodal displacement vector.

This equation has two unknowns $\{R\}$ and $\{U\}$, however, the boundary conditions for displacement constraints in the x and y direction in the truss structure (the base attachment points in the crane design) mean that the displacement at these points is zero. Furthermore, if there are no constraints on a node, the external reaction force will be zero. Using the information from the boundary conditions, the equation can be simplified in a form that eliminates $\{R\}$ and allows for a solution to be found. Then, this solution may be substituted yet again into the initial equation to solve for its value.

Then, the forces on each node and within each member may be solved for by splitting the global matrices back into their corresponding elements as such (note that $\{r\}$ has been excluded):

$$\{f\} = [k]\{u\} \ ,$$

Therefore, the axial forces and thus stresses may be solved in any of the truss elements using the equation above. The explanation for how the truss solver works has been heavily simplified for the sake of brevity, but the mechanisms behind its function and implementation in Python are well defined in various online sources. For further information, please refer to the sources in the footnotes.

Although this algorithm does not have the same precision or complexity as Marc Mentat, it will only be used to obtain a general trend in how the buckling distribution factor varies when the position of nodes changes. Thus, the precise magnitudes of the individual forces or stresses are of limited importance since the primary focus lies in the overarching trend. Therefore, once a final placement has been obtained for each node, Marc Mentat may be used to obtain more accurate predictions for the axial forces themselves.

[3] S. Moaveni, Finite element analysis: Theory and Application with ANSYS. Prentice Hall, 2008.

Objective function

To optimize the truss structure, the value for the buckling distribution factor Γ must be minimized for a set of different node positions. When substituting terms in the expression for Γ in terms of the forces T_i and original lengths $L_{0,i}$ for each element i, the following is obtained:

$$\Gamma = \frac{\sum_{i=0}^{n} T_i \mu_i}{\sum_{i=0}^{n} T_i} + 2 \cdot \sqrt{\frac{\lambda^4}{\alpha^2} \cdot \frac{(\sum_i T_i)(\sum_i T_i^3 L_{0,i}^4) - \left(\sum_i T_i^2 L_{0,i}^2\right)^2}{(\sum_i T_i)^2}}. \tag{4}$$

This expression is quite large and involves multiple complex calculations. For this reason, an 'objective' function is created which takes the inputted lengths of truss members under compression and the magnitude of their corresponding internal forces, and calculates a value for the buckling distribution factor Γ. This function is split into two pieces called `calculate_buckling_indices()` and `get_objective()` which modularizes individual tasks and keeps the code object-oriented. Additionally, the code for `calculate_buckling_indices()`, which calculates the buckling distribution factor Γ, has been included in Appendix B.

Single-point optimization

For a multi-point numerical approach, the case of varying the position of a single point must first be investigated. The aim of single-point variation is to determine the optimal position of a single node in the truss structure by minimizing the value of the buckling distribution factor (calculated using the objective function), while holding all other node positions constant. This will later serve as the foundation for the more complex multi-point optimization.

The process for how this may be done can be described in the following way:

1. A target node is selected for optimization.
2. A two-dimensional grid of potential x and y coordinates is established around the initial position of the node (while ensuring that the coordinates do not enter the exclusion zone, i.e., above 0.9 m and in the region where the wall is located).

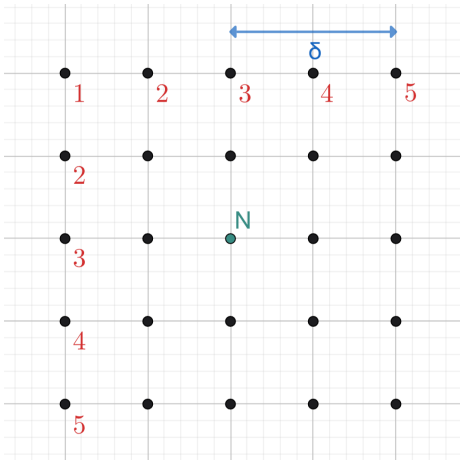

Figure 1: Diagram of two-dimensional variation grid.

3. The two-dimensional grid is split into distinct points within a region δ (or the parameter `delta` in code) units away from the point in the x and y direction, with a number of equally distributed points within this region given by the parameter `grid_points`. Figure 1 above visually shows what this grid of points would look like for a hypothetical target node N and `grid_points`$= 5$ (as the grid is 5×5), with the distance δ labeled.
4. The analysis iterates through each distinct coordinate in this grid. For each pair:

(a) The position of the selected target point is updated within the truss geometry.

(b) The custom Python truss solver is invoked to perform a static analysis of the updated truss and calculate the axial forces in all members.

(c) The function `get_objective()` is invoked and computes the safety margin via `calculate_buckling_indices()` with the solved member loadings and lengths as input parameters.

(d) The computed value from the objective function (or the buckling distribution factor) is stored along with the corresponding node coordinates.

5. The results are plotted as a 3D surface plot, with the x and y axes representing the position of the node and the z-axis indicating the corresponding buckling distribution factor.

Thus, the point with the lowest z-value on the 3D plot corresponds to a point with x and y coordinates that results in the smallest risk of buckling for the truss structure overall. This provides useful insights during the design process of the structure as it allows for a quantitative justification for why certain node coordinates where chosen. To verify whether this algorithm works, Node 6 from Crane Design 3 was selected as the target node, and the algorithm was run.

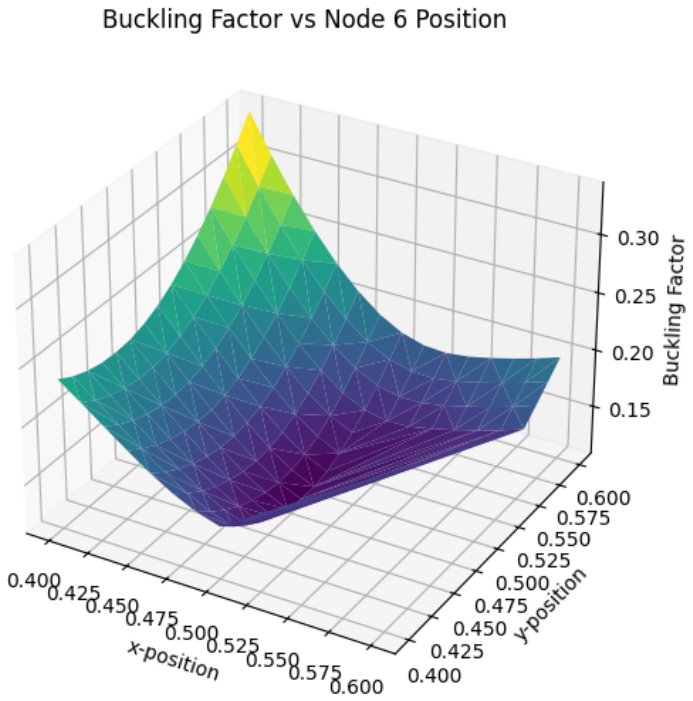

Figure 2: 3D plot of single-point variation for sample point from Crane Design 3.

Figure 2 shows the 3D plot for Node 6 with parameters $\delta = 0.1\,\text{m}$ and `grid_points`$= 15$. It is also important to note how there are points missing in the lower right quarter of the 3D plot, as this region corresponds to the wall, which was set to be excluded from the variation algorithm. Thus, this sample run proves that the algorithm does indeed work for optimizing the position of a single point.

However, this algorithm for single-point variation only allows for the optimization of points one at a time. Hence, an iterative process can be implemented that takes a set of multiple points and optimizes the positions of each point simultaneously.

Multi-point iterative optimization

To approach this problem from a multi-point method, it must first be broken down into smaller pieces. If all nodes are simultaneously varied, the problem becomes computationally intractable. Thus, an iterative approach is utilized in which the single-point optimization is applied to one point while holding the others constant. This process is then repeated numerous times (multiple iterations) to obtain the lowest buckling distribution factor when varying the position of a given set of nodes.

The algorithm for the iterative multi-point optimization can be described as follows:

1. **Initialization:** The process begins with the original truss geometry. A list of all nodes to be optimized is provided.
2. **Iterative passes:** an input is provided for the number of iterative passes, which shall be performed, with a higher number converging toward a more optimal value but requiring more computational time.
3. **Single-point optimization loop:** Within each iteration, the algorithm sequentially optimizes the position of one node at a time:
   (a) The position of all other nodes in the truss is 'frozen' at their current location (which may be the most optimal location from the last iteration).
   (b) The single-point variation function is called for the currently targeted node.
   (c) The minimum buckling distribution factor is identified.
   (d) The corresponding coordinate values are saved, and the position of the target node in the truss structure is updated to the new optimal location.
4. **Convergence:** The iterative approach means that after each iteration, the overall truss structure converges toward the most optimal state.
5. **Optimized structure:** The final optimal positions of the input points are saved, which represent the ideal layout when varying their positions.

By utilizing this iterative multi-point approach, the placement of multiple points may be optimized in the design of the crane. However, it is also important to note that this iterative approach must also be used with the guidance of the user, as simply selecting all unfixed points within the truss structure and running the algorithm can give unstable results in other indices that are not taken into account by this method. Thus, a conservative selection of target points to be optimized shall be made when running this multi-point approach.

Continued on next page

Goal 2 - Application to Crane Design

Subsequently, the iterative multi-point algorithm may be applied to the latest crane design created by the team. For this SSA, Crane Design 3 will be used, which is shown in its original form in the figure below: In this crane design, Node 6 immediately stands out as an area of buckling

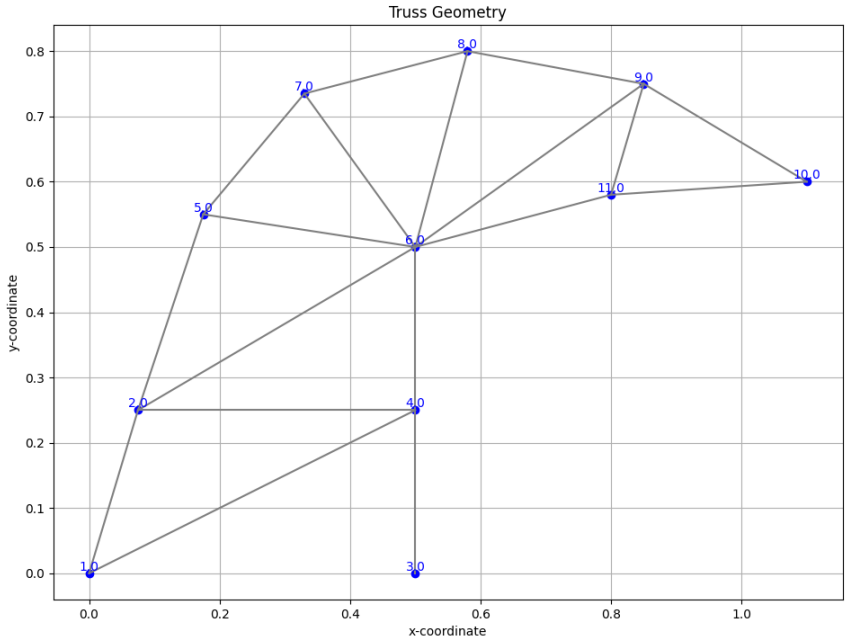

Figure 3: Original Crane Design 3 (before optimization).

risk. The element connecting Nodes 4 and 6 will be under significant compression, and so will the element connecting Nodes 6 and 11. By optimizing the nodes in this region, the design of the crane can be optimized to redistribute the compressive forces by minimizing individual members with significantly higher compression.

In order to determine what parameter values work the best, the algorithm was ran with various combinations of values until an ideal combination was found. This involves a lot of trial-and-error methodology; however, this is often the case for parameter and weight/bias tuning in numerical methods and reinforcement learning.

For this crane design, the points selected for optimization were along the upper left section of the truss. Points much further ahead in the arm were excluded, as well as points in the base, since the algorithm attempted to shift them into unreasonable positions.

After some tuning for the rest of the parameters, the algorithm was run with the following parameters:

- `target_points = [5, 6, 7, 8, 9]`
- `iterations = 6`
- `grid_points = 20`
- `delta = 0.1`

Continued on next page

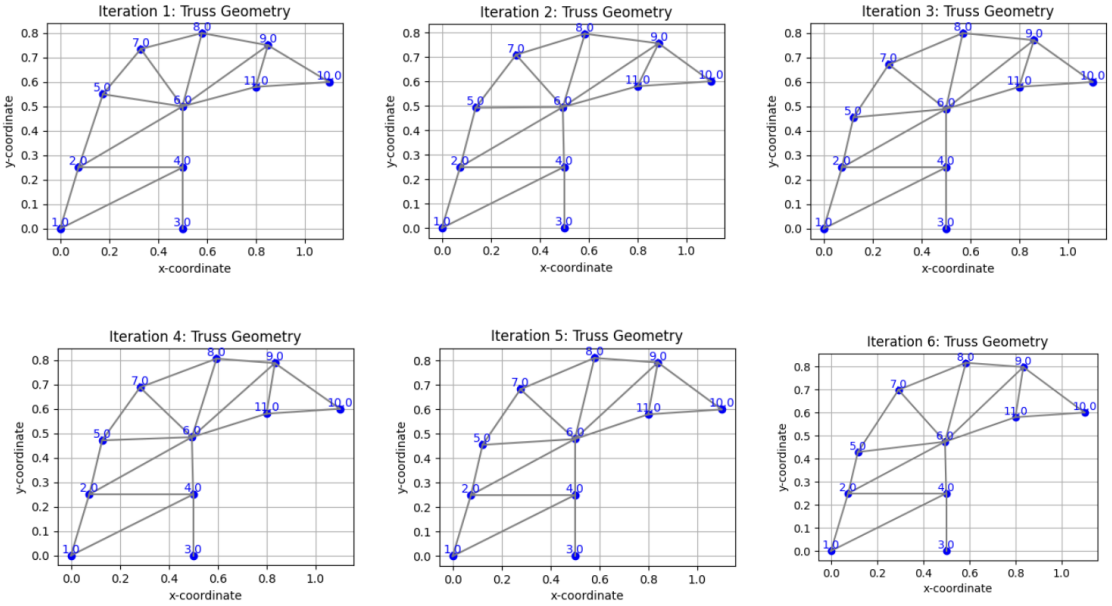

Figure 4: Multi-point algorithm iterations

In Figure 4 above, a representation of how the points shifted after each iterative step. This gives an idea of how each successive iteration converges towards minimizing the objective function (in this case, the buckling distribution factor).

In comparing the first and last iterations, it can be seen that the Nodes begin to spread out– particularly Nodes 5 and 9, which begin to move further away from each other. According to the optimization functions provided, this does indeed reduce the buckling distribution factor. However, it also seems to create a truss that is structurally less stable and likely has increased loadings overall. Therefore, the optimizations made by the algorithm shall be further verified with a final simulation in Marc Mentat.

Goal 3 - Validation of Results

In order to verify whether the current objective function has successfully optimized the crane design, the output points may be entered into a simulation in Marc Mentat to observe the new forces in the truss structure.

The optimized target nodes are as follows (the rest remained the same as they were not passed into the algorithm):

Node	x	y
5	0.122368	0.455263
6	0.500000	0.468421
7	0.298421	0.703421
8	0.580000	0.821053
9	0.807895	0.813158

Table 1: Table of updated target node coordinates

The updated points were then passed into Marc Mentat to perform a structural simulation. It yielded the following results:

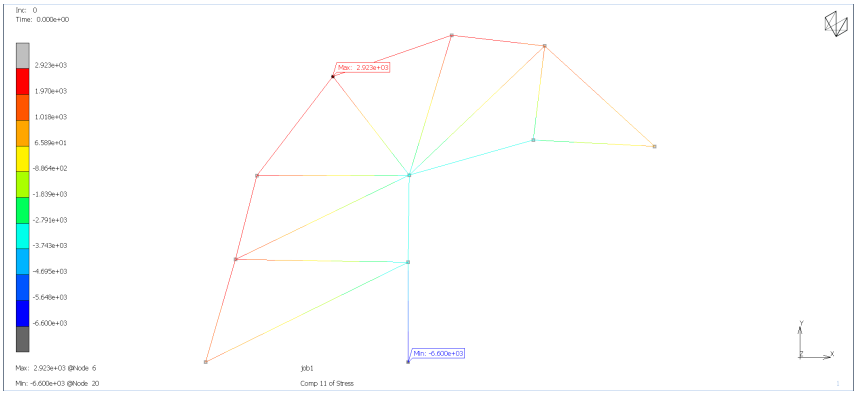

Figure 5: Marc Mentat simulation of optimized crane design.

Despite a reduction in the buckling distribution factor via the multi-point algorithm, the overall forces within the truss structure have increased. However, this does not mean that this entire process cannot be useful. The issue originates from the objective function used for the optimization process, which aimed to only minimize the variation or spread between elements under compression. Therefore, when one element was under significant compression that could not be relieved easily, the algorithm increased the compressive forces within other elements to try to reduce the differences between compressive loads.

Although the buckling distribution factor is important to the truss structure overall, it is only one of many that need to be taken into account. Therefore, a follow-up SSA shall address this via a custom multi-index objective function with various tunable weights placed upon the importance of each. This generalizes the problem and allows for a more thorough numerical optimization process based on a wider variety of factors.

A Source Code for Numerical Method

The source code and instructional information for its use can be found on GitHub at:
https://github.com/madeofcloud/numerical-truss-optimizer

Continued on next page

B Objective Function for Numerical Method

```python
def calculate_buckling_indices(stresses_df, alpha, lambd):
    """
    Calculates the weighted mean, weighted variance, and safety factor
    based on the equations in SSA 3.
    """
    df = stresses_df.copy()

    # Filter for compressive members
    compressive_df = df[df["axial_force"] < 0].copy()

    if compressive_df.empty:
        return {
            "gamma": np.nan,
            "s_mu_sq": np.nan,
            "s_mu": np.nan,
            "safety_margin": np.nan,
        }

    # Calculate utilization ratio mu_i
    # P_c,i = alpha / (lambda^2 * L_0,i^2)
    # mu_i = |T_i / P_c,i| = |axial_force / (alpha / (lambda^2 * L_0,i^2))|
    # L_0,i is the original member length of any element i
    compressive_df['mu'] = np.abs(compressive_df['axial_force']) * (lambd**2 *
    compressive_df['L']**2) / alpha

    # Calculate statistical weight omega_i.
    # We can simplify this for the calculation of gamma and s_mu^2 as shown in the
    derivation
    # in SSA 3, where only T_i remains.
    # The relevant term is the denominator in the final expressions for gamma and
    s_mu^2, which is sum(T_i).
    sum_T = compressive_df['axial_force'].abs().sum()

    if sum_T == 0:
        return {
            "gamma": np.nan,
            "s_mu_sq": np.nan,
            "s_mu": np.nan,
            "safety_margin": np.nan,
        }

    # Calculate weighted mean gamma
    # gamma = sum(T_i * mu_i) / sum(T_i)
    gamma = (compressive_df['axial_force'].abs() * compressive_df['mu']).sum() /
    sum_T

    # Calculate weighted variance s_mu^2
    # s_mu^2 = sum(T_i * (mu_i - gamma)^2) / sum(T_i)
    s_mu_sq = (compressive_df['axial_force'].abs() * (compressive_df['mu'] - gamma)
    **2).sum() / sum_T
    s_mu = np.sqrt(s_mu_sq)

    # Calculate safety margin
    safety_margin = gamma + 2 * s_mu

    return {
        "gamma": gamma,
        "s_mu_sq": s_mu_sq,
        "s_mu": s_mu,
        "safety_margin": safety_margin,
    }
```

Objective function has been created with the assistance of Artificial Intelligence via GitHub Copilot in Microsoft Visual Studio Code.[4]

[4]GitHub, "GitHub Copilot," Microsoft, Accessed: Sep. 9, 2025. [Software]. Available: https://github.com/features/copilot

C Iterative Step for Numerical Method

```python
import numpy as np
from truss_analysis import vary_node_position, get_objective

def run_optimization(data, nodes_to_optimize, n_iterations=2, n_grid_points=5, delta=0.1):
    """Simplified iterative optimization loop."""

    current_data = data.copy()

    for _ in range(n_iterations):
        for node_id in nodes_to_optimize:
            # Get current node position
            x0 = current_data["points"].loc[current_data["points"]['Node'] == node_id, 'x'].values[0]
            y0 = current_data["points"].loc[current_data["points"]['Node'] == node_id, 'y'].values[0]

            # Define search grid around current position
            x_positions = np.linspace(x0 - delta, x0 + delta, n_grid_points)
            y_positions = np.linspace(y0 - delta, y0 + delta, n_grid_points)

            # Evaluate objective function at all grid points
            results_df = vary_node_position(
                current_data,
                node_to_move=node_id,
                x_positions=x_positions,
                y_positions=y_positions,
                objective_fn=get_objective,
                plot=False
            )

            # Update node to best position found
            if not results_df['objective'].isnull().all():
                min_obj_row = results_df.loc[results_df['objective'].idxmin()]
                current_data["points"].loc[current_data["points"]['Node'] == node_id, ['x', 'y']] = [min_obj_row['x'], min_obj_row['y']]

    return current_data
```