ME232 (KDoM) Report

Multi-objective Optimization for Bevel Gears

Group B8

Ayush Kumar Singh (22B2203)

Shahu Patil (22B2146)

Rithvik Subhash (22b2120)

1) Introduction

The main purpose of our coding project was to build an optimizing algorithm for the users such for a given set of the parameters and the variables we will receive the optimized output based upon the objective functions and the constraints defined.

Optimizing the design of a gear pair, particularly a spiral bevel gear pair, offers several benefits to users across various industries.

Optimizing this design is important and can benefit the users in the following ways:

- Enhanced Performance: Optimizing the gear pair design will lead to improved performance characteristics such as higher efficiency, smoother operation, and reduced wear and tear.
- Efficiency Improvement: By maximizing the efficiency of the gear pair, users can achieve better power transmission with minimal energy loss.
- Cost Reduction: A well-optimized gear design will also help to reduce manufacturing costs by minimizing material usage, simplifying production processes, and lowering maintenance requirements.
- Enhanced Durability: Optimal gear designs are less prone to premature failure due to factors such as fatigue, excessive stress, or inadequate lubrication.
- Customization and Adaptability: Optimization allows for the customization of gear designs to meet specific application requirements, such as torque, speed, and space constraints.

Our task involves optimizing the design of a gear pair, specifically a spiral bevel gear pair, to meet certain objectives along with sticking to the specified constraints.

The objectives and the constraints are as follows:

- Objectives:
- Weight Minimization: Second objective is to minimize the total weight of the gear pair. This is essential for applications where weight is a critical factor.

- Efficiency Maximization: The goal is to maximize the efficiency of the gear pair, which is inversely related to the power loss in the system. This involves minimizing factors contributing to power loss such as bending stress, crushing stress, and other geometric factors.
- Pitch Cone Distance Minimization: The pitch cone distance is a measure of the distance between the pitch cones of the gears. Minimizing this distance can lead to better meshing and smoother operation of the gears.

➢ Constraints:

- Bending Stress Constraint: The bending stress on the gear teeth must not exceed a certain allowable limit to prevent failure and fatigue in material.
- Crushing Stress Constraint: Like bending stress, the crushing stress, which is the stress due to compressive forces, must be within the specified range to avoid gear failure.
- Cone Distance Constraint: Due to the limitations on the minimum allowable pitch cone distance to maintain adequate clearance and prevent interference between the gears.
- Average Module Constraint: This constraint ensures that the average module of the gear pair satisfies certain requirements related to shear stress, bending stress, and tooth strength.
- Gear Ratio Constraint: This is an equality constraint specifies a required gear ratio between the two gears in the pair.

The optimization task involves finding the values of design parameters that optimize the objectives while satisfying these constraints. The challenge lies in balancing these conflicting objectives and constraints to arrive at a feasible and optimal gear design.

For this we chose the optimizing algorithm that will take into account these objectives as well as the constraints.

2) Methodology

The general optimization problem looks like:

$$
\begin{aligned}
&\text{Minimize} && (f_1(x), f_2(x), \ldots, f_M(x))^T, \\
&\text{subject to} && g_j(x) \geq 0, \qquad j = 1,2,\ldots,J \\
& && h_k(x) = 0, \qquad k = 1,2,\ldots,K, \\
& && x_i^{(L)} \leq x_i \leq x_i^{(U)}, \qquad i = 1,2,\ldots,n.
\end{aligned}
$$

Here there are m objective function and then there are some of the inequality constraints, equality constraints and then there is also the bounds to the particular variables.

We implemented the NSGA2 algorithm to optimize over the objectives and the constraints.

The objective functions equations are as follows:

a)

Minimisation of weight of the spiral bevel gear pair:

$f_1 = \textit{Total Weight } W = W_1 + W_2$

Where, $W_1 = \textit{Weight of pinion} = 42.438\, \rho m_t^3 z_1$

$W_2 = \textit{Weight of gear} = 68.52\, \rho\, m_t^3 z_2$

Maximisation of efficiency of gear pair:

$f_2 = \textit{Efficiency}\,\eta = 100 - P_L$

Where P_L = Power Loss, which is given by the following equation:

$$50\, f \left\{ \frac{\cos + \cos}{\cos \phi_n} \right\} \cos^2 \beta \, \frac{(H_s^2 + H_t^2)}{(H_s + H_t)}$$

To calculate H_s and H_t, following equations (8) and (9) are used

$$H_S = (i+1) \left\{ \left[\sqrt{\left(\frac{R_o}{R}\right)^2 - \cos^2 \varnothing_n} \right] - \sin \varnothing_n \right\}$$

$$H_t = \left(\frac{i+1}{i}\right) \left\{ \left[\sqrt{\left(\frac{r_o}{r}\right)^2 - \cos^2 \varnothing_n} \right] - \sin \varnothing_n \right\}$$

$$R_o = R + \textit{one addendum}$$

One addendum for 20° full depth involute system = *One average Module* $= m_{av}$

Where,

$$m_{av} = m_t \left(\frac{\Psi_y - 0.5}{\Psi_y} \right)$$

Also,

$r_o = r + m_{av}$; $R_o = R + m_{av}$; $r = d_1/2$ & $R = d_1/2$

Where,

d_1 = Pitch diameter of the large end of bevel pinion in mm = $m_t z_1$

d_2 = Pitch diameter of the large end of bevel gear in mm = $m_t z_2$

b)

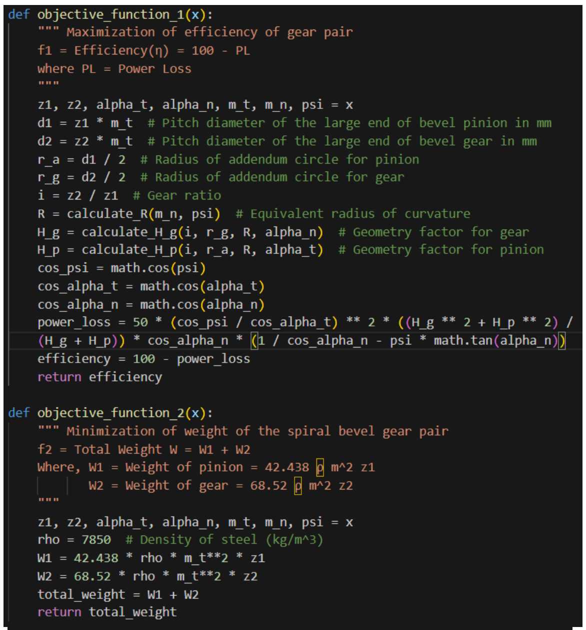
Minimisation of pitch cone distance of gear pair:

Eqn. 10 represents this objective function.

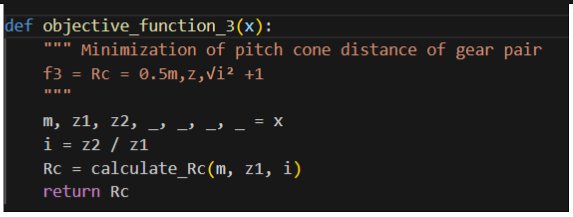
$$f3 = Rc = 0.5 m_t z_1 \sqrt{i^2 + 1}$$

c)

The respective python codes to implement these objectives are shown besides the equations.



```python
def objective_function_1(x):
    """ Maximization of efficiency of gear pair
    f1 = Efficiency(η) = 100 - PL
    where PL = Power Loss
    """
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    d1 = z1 * m_t  # Pitch diameter of the large end of bevel pinion in mm
    d2 = z2 * m_t  # Pitch diameter of the large end of bevel gear in mm
    r_a = d1 / 2  # Radius of addendum circle for pinion
    r_g = d2 / 2  # Radius of addendum circle for gear
    i = z2 / z1  # Gear ratio
    R = calculate_R(m_n, psi)  # Equivalent radius of curvature
    H_g = calculate_H_g(i, r_g, R, alpha_n)  # Geometry factor for gear
    H_p = calculate_H_p(i, r_a, R, alpha_t)  # Geometry factor for pinion
    cos_psi = math.cos(psi)
    cos_alpha_t = math.cos(alpha_t)
    cos_alpha_n = math.cos(alpha_n)
    power_loss = 50 * (cos_psi / cos_alpha_t) ** 2 * ((H_g ** 2 + H_p ** 2) /
    (H_g + H_p)) * cos_alpha_n * (1 / cos_alpha_n - psi * math.tan(alpha_n))
    efficiency = 100 - power_loss
    return efficiency

def objective_function_2(x):
    """ Minimization of weight of the spiral bevel gear pair
    f2 = Total Weight W = W1 + W2
    Where, W1 = Weight of pinion = 42.438 ρ m^2 z1
           W2 = Weight of gear = 68.52 ρ m^2 z2
    """
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    rho = 7850  # Density of steel (kg/m^3)
    W1 = 42.438 * rho * m_t**2 * z1
    W2 = 68.52 * rho * m_t**2 * z2
    total_weight = W1 + W2
    return total_weight
```



```python
def objective_function_3(x):
    """ Minimization of pitch cone distance of gear pair
    f3 = Rc = 0.5m,z,√i² +1
    """
    m, z1, z2, _, _, _, _ = x
    i = z2 / z1
    Rc = calculate_Rc(m, z1, i)
    return Rc
```

The constraints that were needed to prevent the failure and other properties are:

a) Bending stress

$$\sigma_b \le [\sigma_b]$$

$$\sigma_b = \left(\frac{0.7R\sqrt{(i^2+1)}[M_t]}{(R-0.5b)^2 \, bm_n y_v} \right)$$

b) Crushing stress

$$\sigma_c \le [\sigma_c]$$

$$\sigma_c = \frac{0.72}{(R-0.5b)} \sqrt{\frac{(i^2 \pm 1)^3}{ib} E \, [M_t]}$$

c) Cone distance

$$R_{\min} \le R$$

$$\frac{41.4885}{(0.357Z_1 - 0.5)^{\frac{2}{3}}} \le R$$

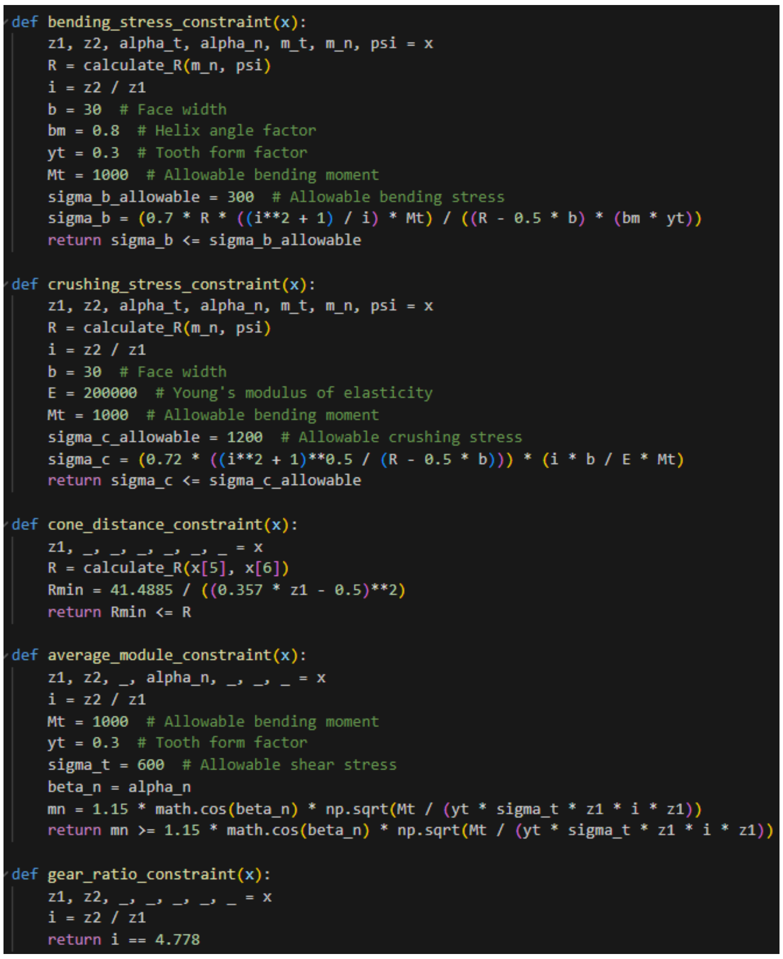
d) Average Module

$$m_{av} \ge 1.15 \cos\beta_{av} \sqrt[3]{\frac{[M_t]}{y_{v[\sigma_b]\psi_m Z_1}}}$$

e) Gear ratio

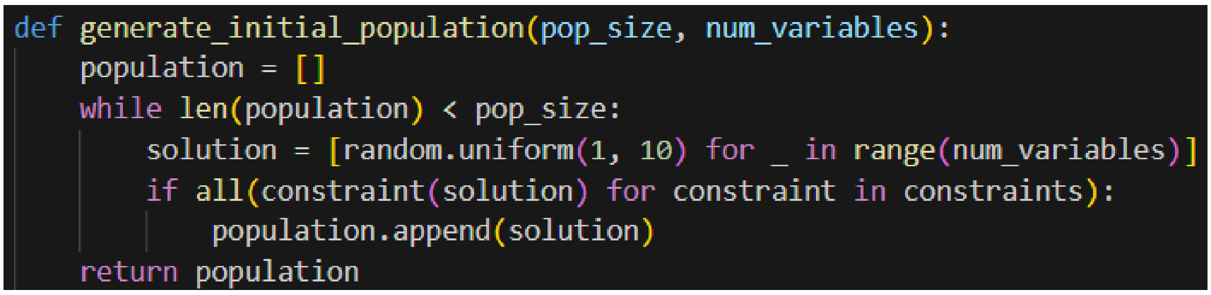
$$i = 4.778$$

Gear Ratio is kept constant



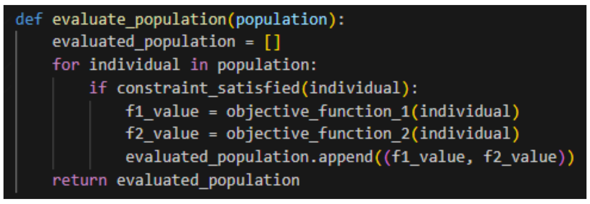
```python
def bending_stress_constraint(x):
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    R = calculate_R(m_n, psi)
    i = z2 / z1
    b = 30  # Face width
    bm = 0.8  # Helix angle factor
    yt = 0.3  # Tooth form factor
    Mt = 1000  # Allowable bending moment
    sigma_b_allowable = 300  # Allowable bending stress
    sigma_b = (0.7 * R * ((i**2 + 1) / i) * Mt) / ((R - 0.5 * b) * (bm * yt))
    return sigma_b <= sigma_b_allowable

def crushing_stress_constraint(x):
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    R = calculate_R(m_n, psi)
    i = z2 / z1
    b = 30  # Face width
    E = 200000  # Young's modulus of elasticity
    Mt = 1000  # Allowable bending moment
    sigma_c_allowable = 1200  # Allowable crushing stress
    sigma_c = (0.72 * ((i**2 + 1)**0.5 / (R - 0.5 * b))) * (i * b / E * Mt)
    return sigma_c <= sigma_c_allowable

def cone_distance_constraint(x):
    z1, _, _, _, _, _, _ = x
    R = calculate_R(x[5], x[6])
    Rmin = 41.4885 / ((0.357 * z1 - 0.5)**2)
    return Rmin <= R

def average_module_constraint(x):
    z1, z2, _, alpha_n, _, _, _ = x
    i = z2 / z1
    Mt = 1000  # Allowable bending moment
    yt = 0.3  # Tooth form factor
    sigma_t = 600  # Allowable shear stress
    beta_n = alpha_n
    mn = 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))
    return mn >= 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))

def gear_ratio_constraint(x):
    z1, z2, _, _, _, _, _ = x
    i = z2 / z1
    return i == 4.778
```

- NSGA2 algorithm starts by initializing the population randomly and then for each individual in the population, random values within specified ranges are assigned which represents potential solutions to the optimization problem.

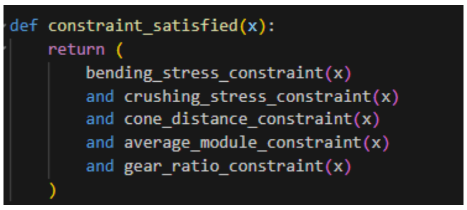


```python
def generate_initial_population(pop_size, num_variables):
    population = []
    while len(population) < pop_size:
        solution = [random.uniform(1, 10) for _ in range(num_variables)]
        if all(constraint(solution) for constraint in constraints):
            population.append(solution)
    return population
```

- Evaluation of the Objective Functions: This function evaluates each of the individual in the population using the objective functions and the given constraints.

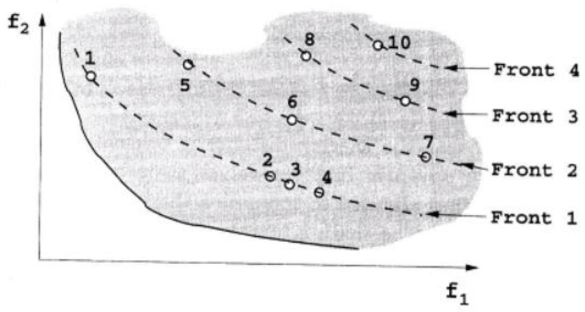


```python
def evaluate_population(population):
    evaluated_population = []
    for individual in population:
        if constraint_satisfied(individual):
            f1_value = objective_function_1(individual)
            f2_value = objective_function_2(individual)
            evaluated_population.append((f1_value, f2_value))
    return evaluated_population
```

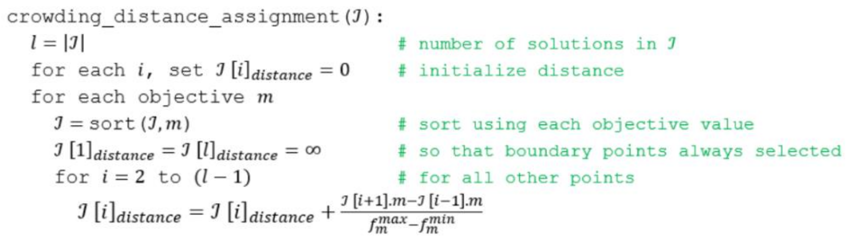


```python
def constraint_satisfied(x):
    return (
        bending_stress_constraint(x)
        and crushing_stress_constraint(x)
        and cone_distance_constraint(x)
        and average_module_constraint(x)
        and gear_ratio_constraint(x)
    )
```

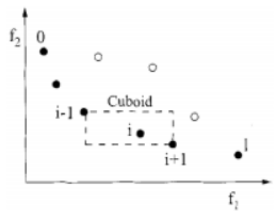
- Dominance Relationship: An individual is said to dominate another individual if any other data point is better than it for any of the given objective functions i.e. in at least one objective and not worse in any objective.
  Below is the more visual explanation of the ranks which are assigned as per the fronts to the individuals.



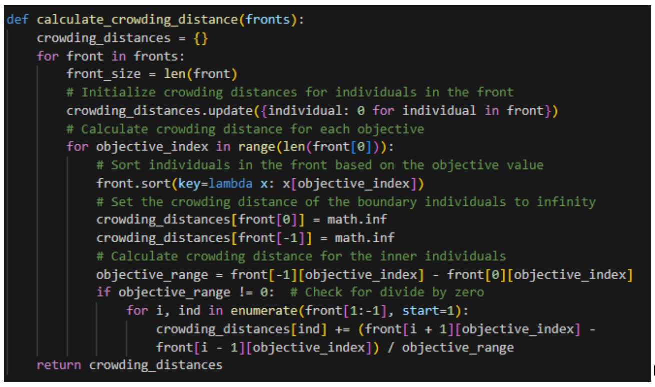
- Crowding Distance Calculation: The crowding distance for individuals is calculated in each front to maintain diversity and to prioritize solutions closer to the Pareto front.
  The extreme most points in the front are allotted the very higher value of the crowding distance value and then the crowding distance for the rest of the individuals in the front are calculated as per:



```
crowding_distance_assignment($\mathcal{I}$):
  $l = |\mathcal{I}|$                                   # number of solutions in $\mathcal{I}$
  for each $i$, set $\mathcal{I}[i]_{distance} = 0$      # initialize distance
  for each objective $m$
    $\mathcal{I} = \text{sort}(\mathcal{I}, m)$          # sort using each objective value
    $\mathcal{I}[1]_{distance} = \mathcal{I}[l]_{distance} = \infty$   # so that boundary points always selected
    for $i = 2$ to $(l-1)$                              # for all other points
      $\mathcal{I}[i]_{distance} = \mathcal{I}[i]_{distance} + \frac{\mathcal{I}[i+1].m - \mathcal{I}[i-1].m}{f_m^{max} - f_m^{min}}$
```



Hypercuboid



```python
def calculate_crowding_distance(fronts):
    crowding_distances = {}
    for front in fronts:
        front_size = len(front)
        # Initialize crowding distances for individuals in the front
        crowding_distances.update({individual: 0 for individual in front})
        # Calculate crowding distance for each objective
        for objective_index in range(len(front[0])):
            # Sort individuals in the front based on the objective value
            front.sort(key=lambda x: x[objective_index])
            # Set the crowding distance of the boundary individuals to infinity
            crowding_distances[front[0]] = math.inf
            crowding_distances[front[-1]] = math.inf
            # Calculate crowding distance for the inner individuals
            objective_range = front[-1][objective_index] - front[0][objective_index]
            if objective_range != 0:  # Check for divide by zero
                for i, ind in enumerate(front[1:-1], start=1):
                    crowding_distances[ind] += (front[i + 1][objective_index] -
                    front[i - 1][objective_index]) / objective_range
    return crowding_distances
```

Code of crowding distance

- Elitism in Selection:

  Identifying the Best Solutions:

  We categorized the individuals into different fronts based on Pareto dominance. Fronts closer to the Pareto front contain better solutions, as they are not dominated by any other individuals in the population. Within each front, individuals were assigned a crowding distance. Individuals with larger crowding distances contribute to maintaining diversity within the population.

  Preservation of Best Solutions:

  Elitism ensured that a certain portion of the best individuals, basically those from the first front are directly carried over to the next generation without undergoing genetic operations such as crossover and mutation.

  By preserving the best solutions, the algorithm prevented the loss of high-quality solutions and maintained the progress towards finding the Pareto-optimal front. It also allowed the introduction of new solutions through genetic operators applied to the rest of the population.
  This combination of preserving elite solutions and generating new offspring helps balance the exploitation of known good solutions with the exploration of new regions in the solution space.

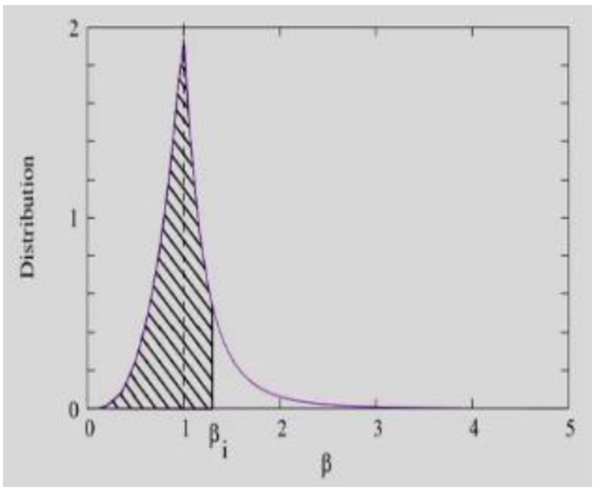
  Next Generation Formation:

  After selecting elite individuals, the remaining slots in the next generation were filled by offspring generated through crossover and mutation operations.
  In order to explore the sample space and generate the diverse population of the of the candidate solutions genetic operators were used.

- Crossover Operator:

  Exploration of Solution Space: By combining different parts of parent solutions, crossover creates offspring that inherit characteristics from multiple parents. This process allows the algorithm to explore new regions of the solution space that may contain promising solutions not present in the current population.

$$p(\beta_i) = \begin{cases} 0.5(\eta_c + 1)\beta_i^{\eta_c}, & if\ \beta_i \le 1 \\ 0.5(\eta_c + 1)\dfrac{1}{\beta_i^{\eta_c+2}}, & otherwise \end{cases}$$



$$\beta_i = \begin{cases} (2u_i)^{\frac{1}{\eta_c+1}} & if\ u_i \le 0.5 \\ \left(\dfrac{1}{2(1-u_i)}\right)^{\frac{1}{\eta_c+1}} & otherwise \end{cases}$$

$$x_i^{(1,t+1)} = 0.5\left[\left(x_i^{(1,t)} + x_i^{(2,t)}\right) - \beta_i\left(x_i^{(2,t)} - x_i^{(1,t)}\right)\right]$$

$$x_i^{(2,t+1)} = 0.5\left[\left(x_i^{(1,t)} + x_i^{(2,t)}\right) + \beta_i\left(x_i^{(2,t)} - x_i^{(1,t)}\right)\right]$$

- Mutation Operator:
  It introduces randomness into the population by making small, random changes to individual solutions. It helps prevent premature convergence by maintaining genetic diversity within the population.

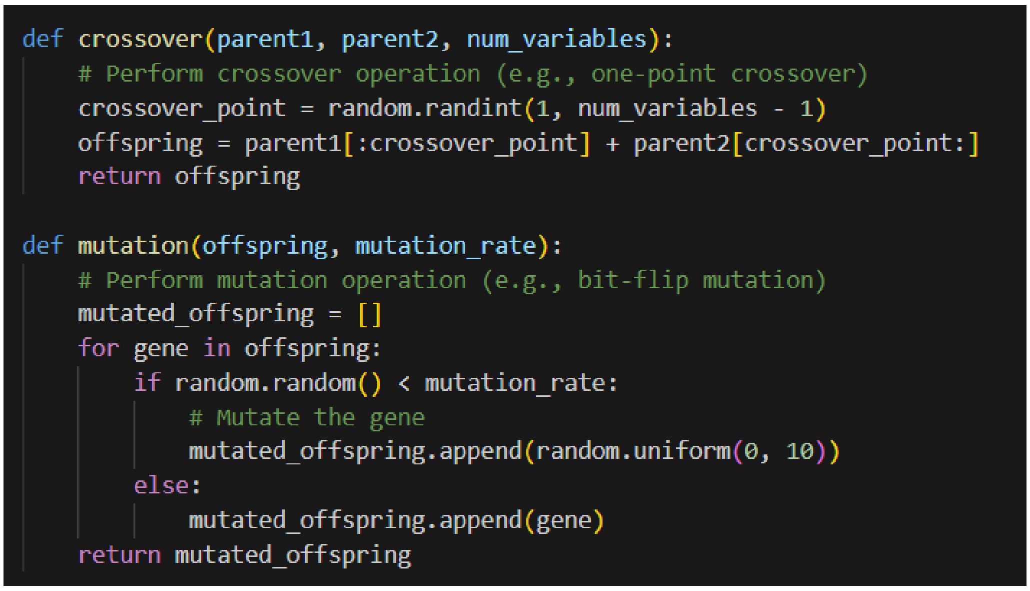
  Without mutation, the population may converge prematurely to a suboptimal solution or become trapped in local optima. Mutation helps disrupt this convergence by introducing variation, allowing the algorithm to continue exploring the solution space and potentially find better solutions.

$$y_i^{(1,t+1)} = x_i^{(1,t+1)} + (x_i^{(U)} - x_i^{(L)})\bar{\delta}_i \qquad P(\delta) = 0.5(\eta_m + 1)(1 - |\delta|)^{\eta_m} :$$

$$\bar{\delta}_i = \begin{cases} (2r_i)^{\frac{1}{\eta_m+1}} - 1, & if\ r_i < 0.5 \\ 1 - [2(1-r_i)]^{\frac{1}{\eta_m+1}}, & if\ r_i \ge 0.5 \end{cases}$$

Rate Control: The mutation rate determines the likelihood of mutation occurring in each individual. A low mutation rate ensures that only a small proportion of individuals undergo mutation, preventing excessive disruption to the population, while a higher mutation rate allows for more extensive exploration of the solution space.

In conclusion, crossover combines genetic material from parent solutions to create offspring, while mutation introduces random changes to individual solutions, ensuring that the algorithm can explore new regions of the solution space and avoid premature convergence.

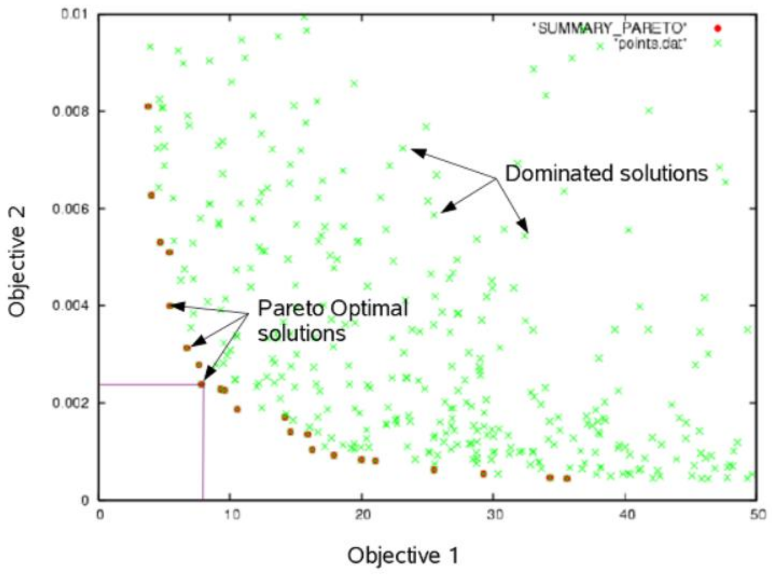


```python
def crossover(parent1, parent2, num_variables):
    # Perform crossover operation (e.g., one-point crossover)
    crossover_point = random.randint(1, num_variables - 1)
    offspring = parent1[:crossover_point] + parent2[crossover_point:]
    return offspring

def mutation(offspring, mutation_rate):
    # Perform mutation operation (e.g., bit-flip mutation)
    mutated_offspring = []
    for gene in offspring:
        if random.random() < mutation_rate:
            # Mutate the gene
            mutated_offspring.append(random.uniform(0, 10))
        else:
            mutated_offspring.append(gene)
    return mutated_offspring
```

- Survivor Selection:
  It is the process of determining which individuals from the parent and offspring populations will proceed to the next generation.
  The environmental survival aspect of survivor selection involves combining the parent and offspring populations and selecting individuals for the next generation based on non-dominated sorting and crowding distance. This ensures that the fittest individuals, in terms of being non-dominated and well-distributed across the objective space, survive to the next generation.

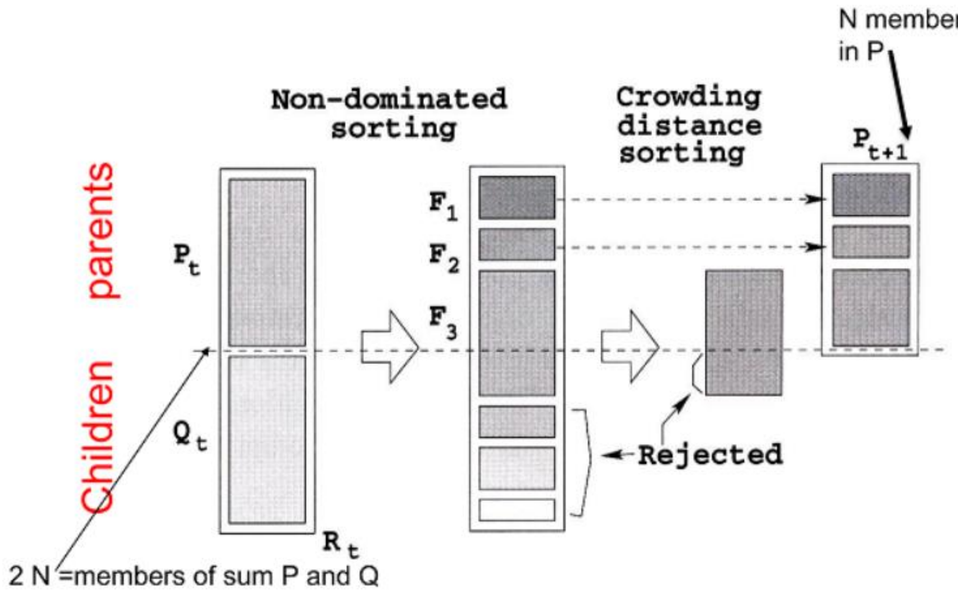
- NSGA-II Implementation:
  Algorithm Parameters: Key parameters such as population size N, number of objectives m, maximum generations max_G and genetic operator probabilities are set before running the NSGA-II algorithm.

  Pareto-optimal Solutions: The Pareto-optimal solutions provided insights into the trade-offs between conflicting objectives. It helps to identify the set of optimal solutions that represent the best compromises between different objectives.

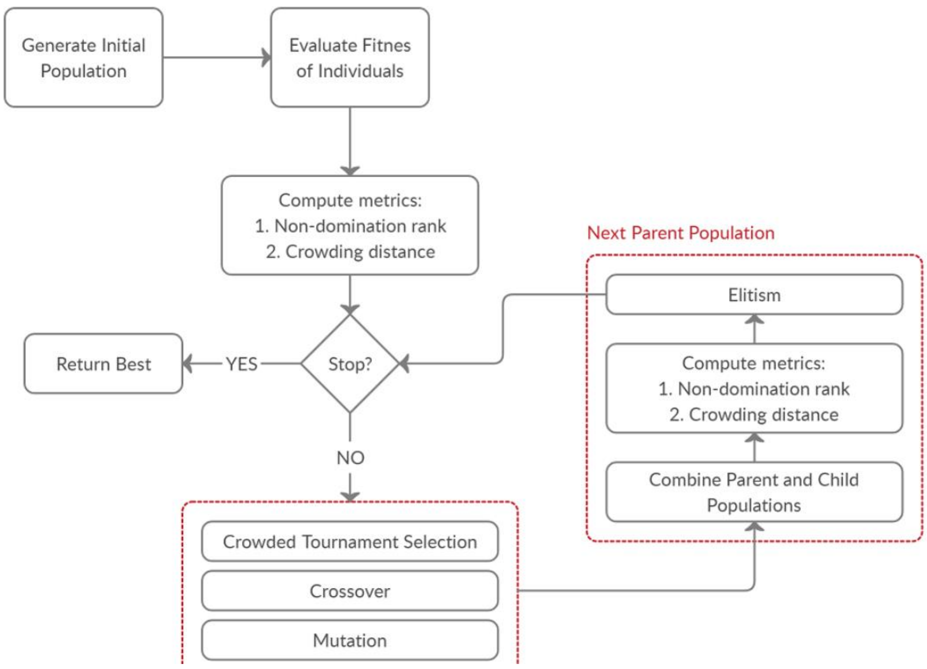


Robustness Analysis: Sensitivity analysis was done to evaluate the robustness of the optimized solutions concerning variations in design parameters and constraints. This analysis helps assess the reliability and stability of the solutions under different scenarios or uncertainties.

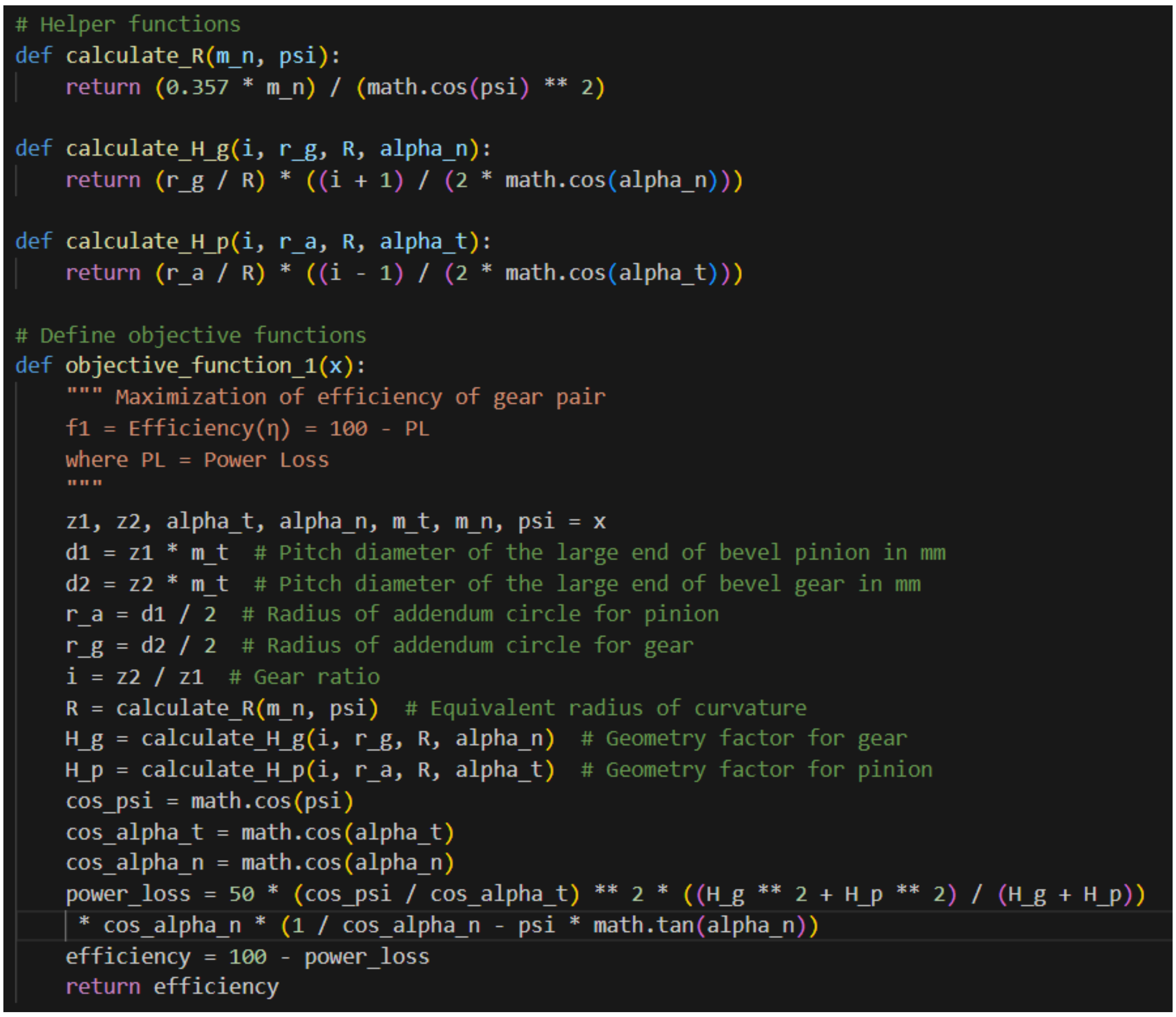
The selection and the rejection of the individuals on the basis of the operators and the ranking and crowding distance.



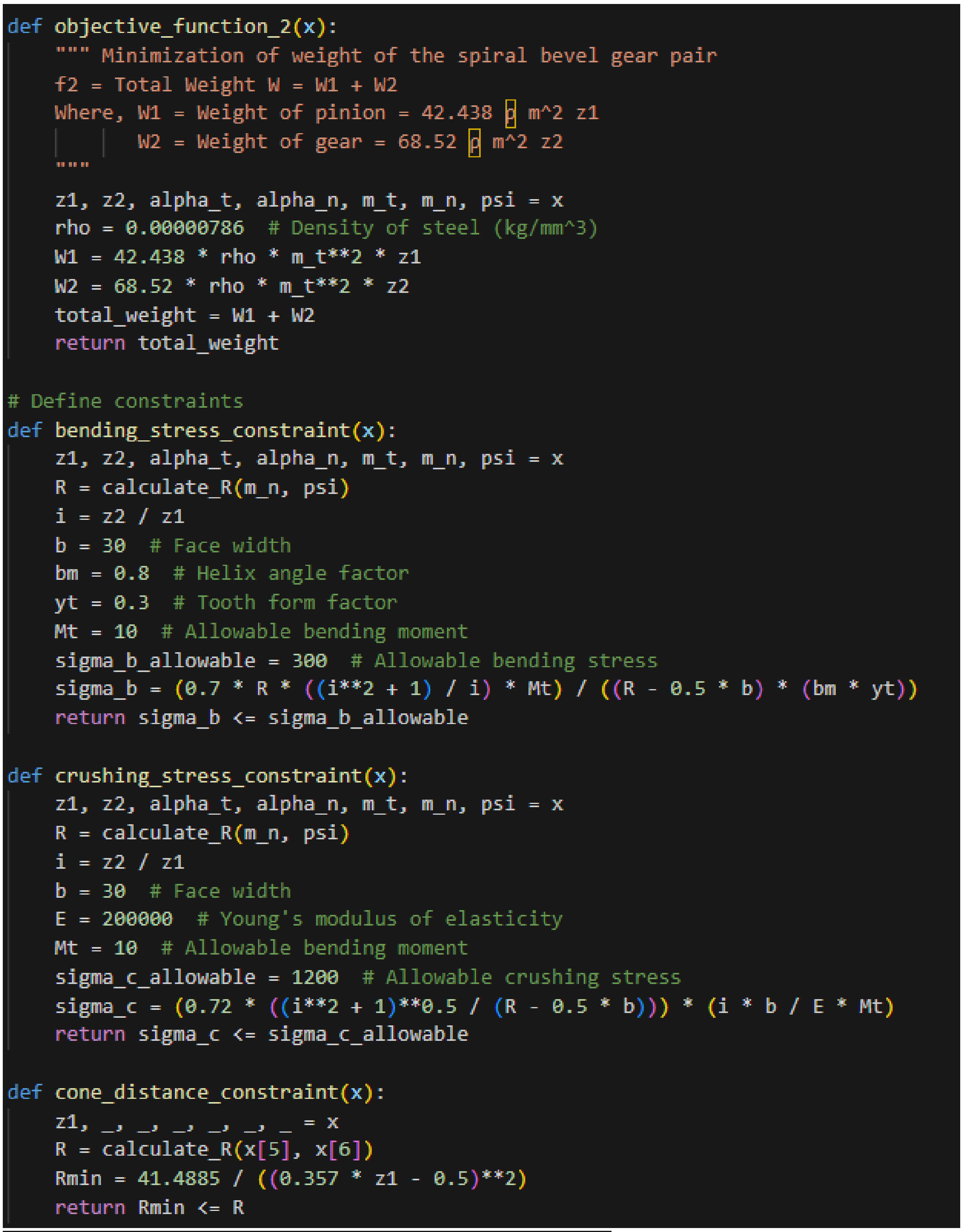
The full flow chart of the implementation of the NSGA2 algorithms is shown on the next page:



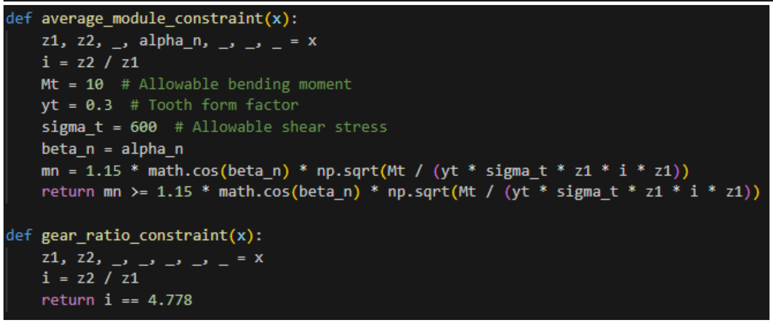
3) Implementation



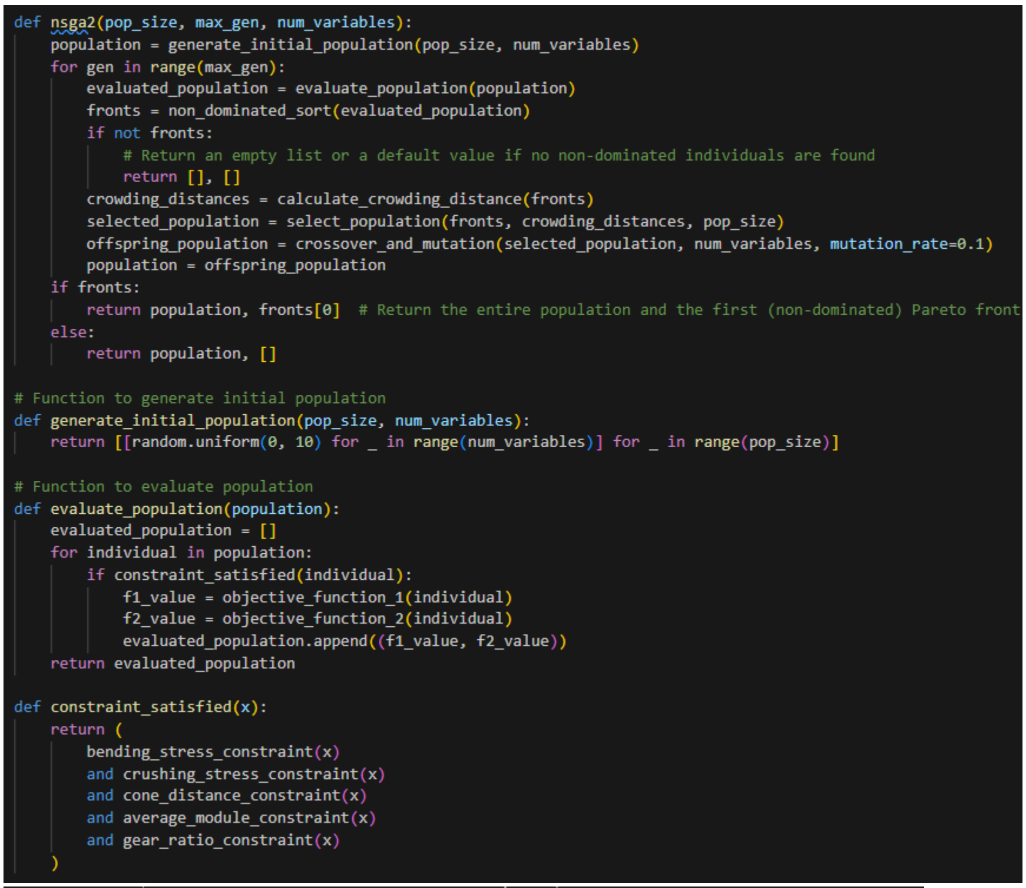
```python
# Helper functions
def calculate_R(m_n, psi):
    return (0.357 * m_n) / (math.cos(psi) ** 2)

def calculate_H_g(i, r_g, R, alpha_n):
    return (r_g / R) * ((i + 1) / (2 * math.cos(alpha_n)))

def calculate_H_p(i, r_a, R, alpha_t):
    return (r_a / R) * ((i - 1) / (2 * math.cos(alpha_t)))

# Define objective functions
def objective_function_1(x):
    """ Maximization of efficiency of gear pair
    f1 = Efficiency(η) = 100 - PL
    where PL = Power Loss
    """
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    d1 = z1 * m_t  # Pitch diameter of the large end of bevel pinion in mm
    d2 = z2 * m_t  # Pitch diameter of the large end of bevel gear in mm
    r_a = d1 / 2  # Radius of addendum circle for pinion
    r_g = d2 / 2  # Radius of addendum circle for gear
    i = z2 / z1  # Gear ratio
    R = calculate_R(m_n, psi)  # Equivalent radius of curvature
    H_g = calculate_H_g(i, r_g, R, alpha_n)  # Geometry factor for gear
    H_p = calculate_H_p(i, r_a, R, alpha_t)  # Geometry factor for pinion
    cos_psi = math.cos(psi)
    cos_alpha_t = math.cos(alpha_t)
    cos_alpha_n = math.cos(alpha_n)
    power_loss = 50 * (cos_psi / cos_alpha_t) ** 2 * ((H_g ** 2 + H_p ** 2) / (H_g + H_p))
     * cos_alpha_n * (1 / cos_alpha_n - psi * math.tan(alpha_n))
    efficiency = 100 - power_loss
    return efficiency
```



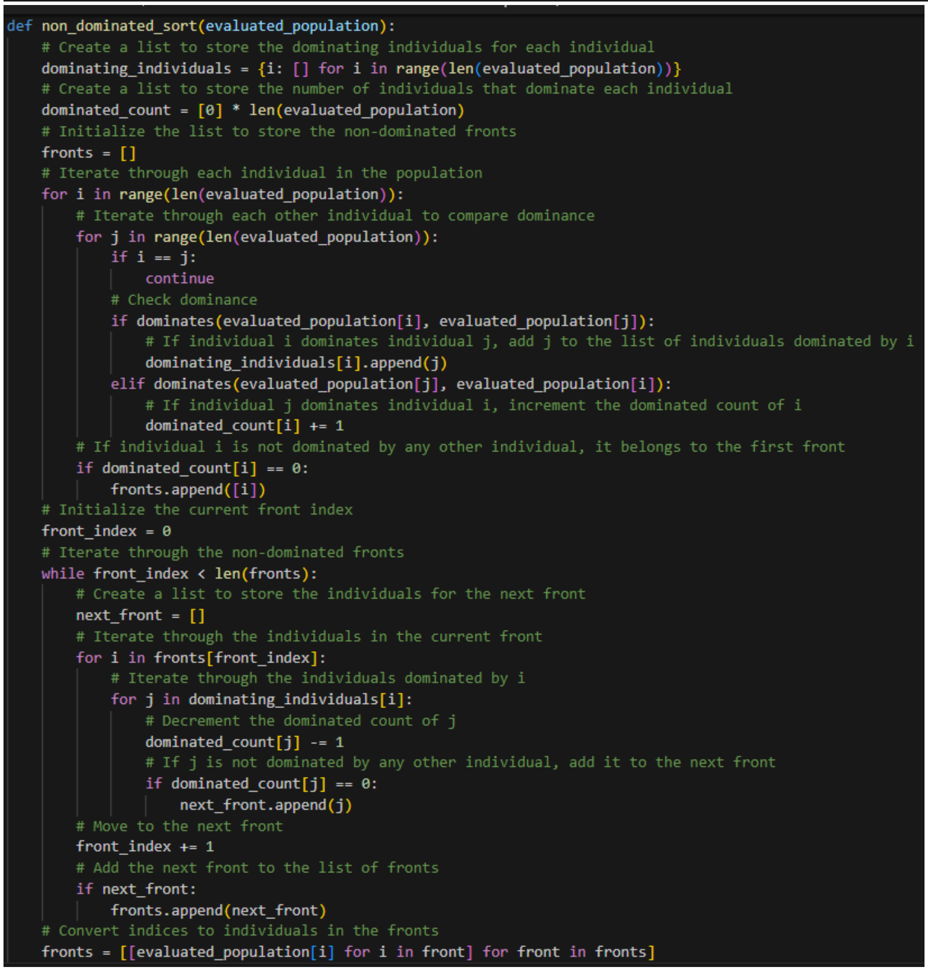
```python
def objective_function_2(x):
    """ Minimization of weight of the spiral bevel gear pair
    f2 = Total Weight W = W1 + W2
    Where, W1 = Weight of pinion = 42.438 ρ m^2 z1
              W2 = Weight of gear = 68.52 ρ m^2 z2
    """
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    rho = 0.00000786  # Density of steel (kg/mm^3)
    W1 = 42.438 * rho * m_t**2 * z1
    W2 = 68.52 * rho * m_t**2 * z2
    total_weight = W1 + W2
    return total_weight

# Define constraints
def bending_stress_constraint(x):
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    R = calculate_R(m_n, psi)
    i = z2 / z1
    b = 30  # Face width
    bm = 0.8  # Helix angle factor
    yt = 0.3  # Tooth form factor
    Mt = 10  # Allowable bending moment
    sigma_b_allowable = 300  # Allowable bending stress
    sigma_b = (0.7 * R * ((i**2 + 1) / i) * Mt) / ((R - 0.5 * b) * (bm * yt))
    return sigma_b <= sigma_b_allowable

def crushing_stress_constraint(x):
    z1, z2, alpha_t, alpha_n, m_t, m_n, psi = x
    R = calculate_R(m_n, psi)
    i = z2 / z1
    b = 30  # Face width
    E = 200000  # Young's modulus of elasticity
    Mt = 10  # Allowable bending moment
    sigma_c_allowable = 1200  # Allowable crushing stress
    sigma_c = (0.72 * ((i**2 + 1)**0.5 / (R - 0.5 * b))) * (i * b / E * Mt)
    return sigma_c <= sigma_c_allowable

def cone_distance_constraint(x):
    z1, _, _, _, _, _, _ = x
    R = calculate_R(x[5], x[6])
    Rmin = 41.4885 / ((0.357 * z1 - 0.5)**2)
    return Rmin <= R
```



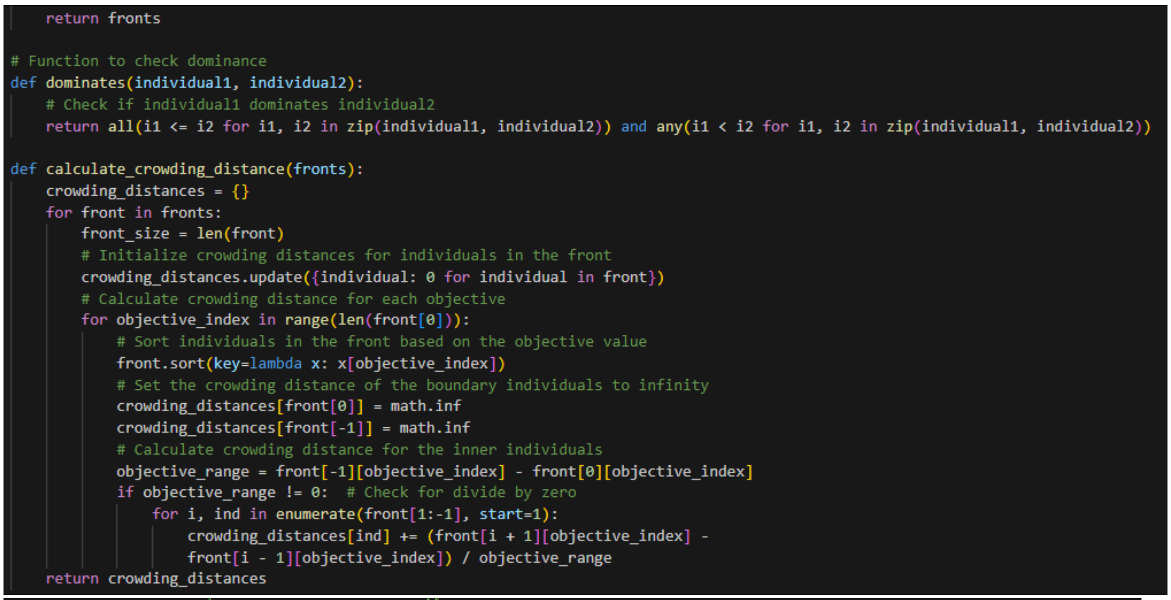
```python
def average_module_constraint(x):
    z1, z2, _, alpha_n, _, _, _ = x
    i = z2 / z1
    Mt = 10  # Allowable bending moment
    yt = 0.3  # Tooth form factor
    sigma_t = 600  # Allowable shear stress
    beta_n = alpha_n
    mn = 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))
    return mn >= 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))

def gear_ratio_constraint(x):
    z1, z2, _, _, _, _, _ = x
    i = z2 / z1
    return i == 4.778
```



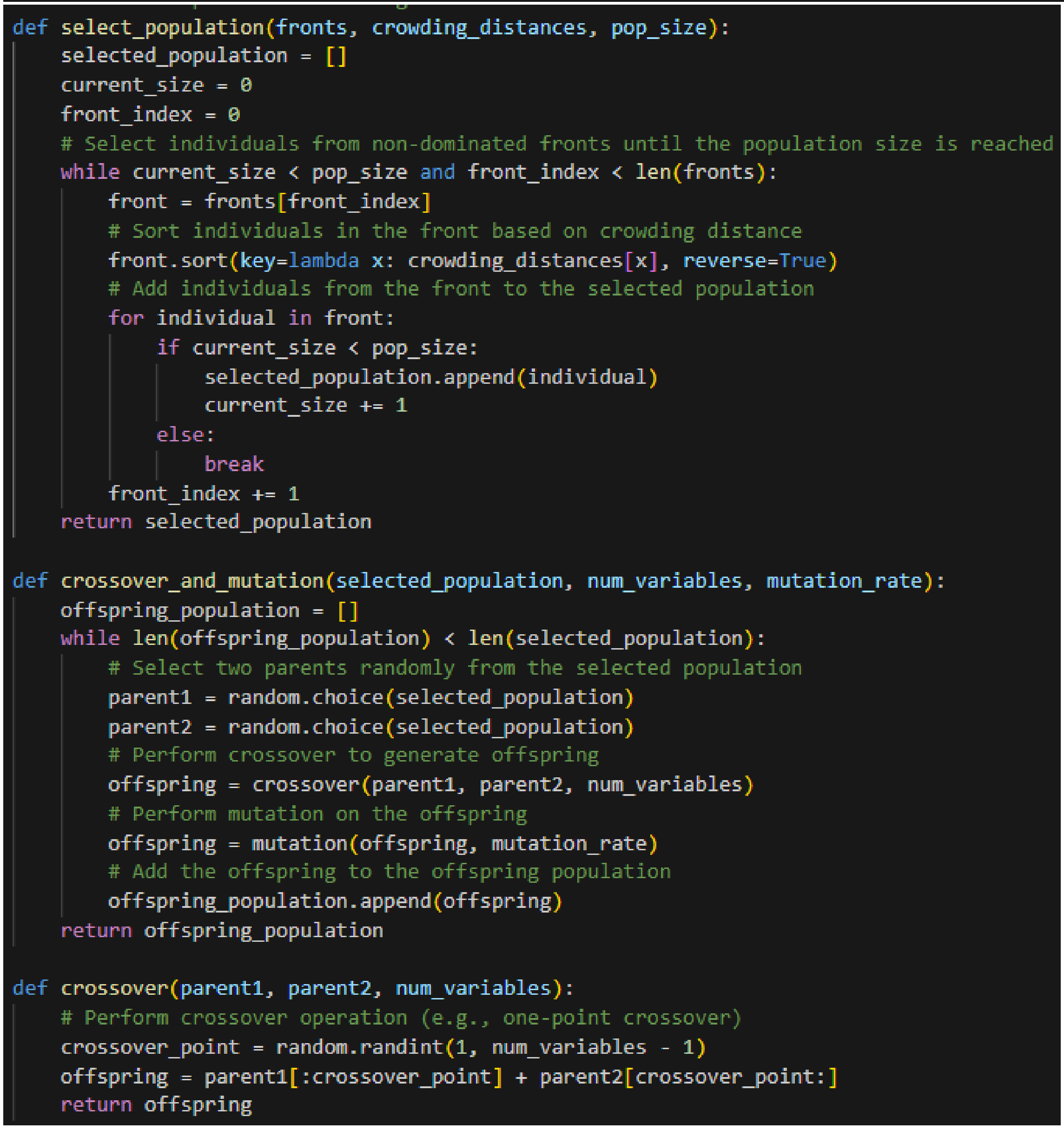
```
def nsga2(pop_size, max_gen, num_variables):
    population = generate_initial_population(pop_size, num_variables)
    for gen in range(max_gen):
        evaluated_population = evaluate_population(population)
        fronts = non_dominated_sort(evaluated_population)
        if not fronts:
            # Return an empty list or a default value if no non-dominated individuals are found
            return [], []
        crowding_distances = calculate_crowding_distance(fronts)
        selected_population = select_population(fronts, crowding_distances, pop_size)
        offspring_population = crossover_and_mutation(selected_population, num_variables, mutation_rate=0.1)
        population = offspring_population
    if fronts:
        return population, fronts[0]  # Return the entire population and the first (non-dominated) Pareto front
    else:
        return population, []

# Function to generate initial population
def generate_initial_population(pop_size, num_variables):
    return [[random.uniform(0, 10) for _ in range(num_variables)] for _ in range(pop_size)]

# Function to evaluate population
def evaluate_population(population):
    evaluated_population = []
    for individual in population:
        if constraint_satisfied(individual):
            f1_value = objective_function_1(individual)
            f2_value = objective_function_2(individual)
            evaluated_population.append((f1_value, f2_value))
    return evaluated_population

def constraint_satisfied(x):
    return (
        bending_stress_constraint(x)
        and crushing_stress_constraint(x)
        and cone_distance_constraint(x)
        and average_module_constraint(x)
        and gear_ratio_constraint(x)
    )
```



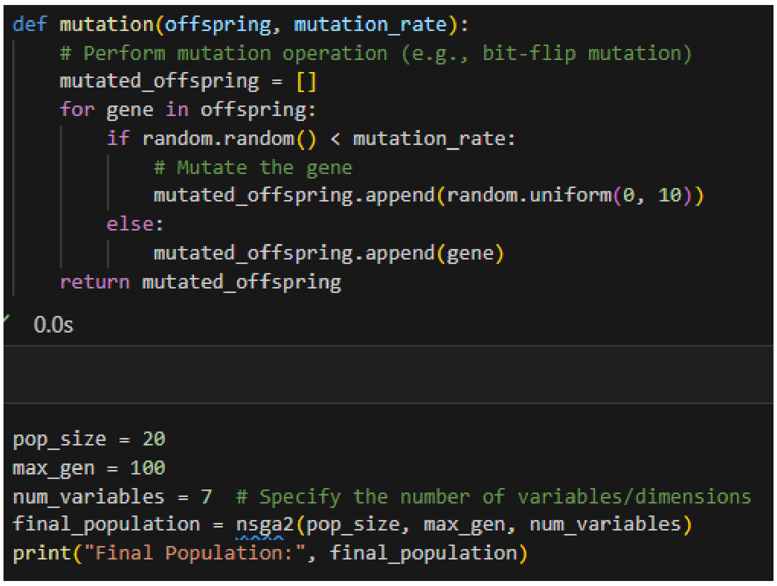
```
def non_dominated_sort(evaluated_population):
    # Create a list to store the dominating individuals for each individual
    dominating_individuals = {i: [] for i in range(len(evaluated_population))}
    # Create a list to store the number of individuals that dominate each individual
    dominated_count = [0] * len(evaluated_population)
    # Initialize the list to store the non-dominated fronts
    fronts = []
    # Iterate through each individual in the population
    for i in range(len(evaluated_population)):
        # Iterate through each other individual to compare dominance
        for j in range(len(evaluated_population)):
            if i == j:
                continue
            # Check dominance
            if dominates(evaluated_population[i], evaluated_population[j]):
                # If individual i dominates individual j, add j to the list of individuals dominated by i
                dominating_individuals[i].append(j)
            elif dominates(evaluated_population[j], evaluated_population[i]):
                # If individual j dominates individual i, increment the dominated count of i
                dominated_count[i] += 1
        # If individual i is not dominated by any other individual, it belongs to the first front
        if dominated_count[i] == 0:
            fronts.append([i])
    # Initialize the current front index
    front_index = 0
    # Iterate through the non-dominated fronts
    while front_index < len(fronts):
        # Create a list to store the individuals for the next front
        next_front = []
        # Iterate through the individuals in the current front
        for i in fronts[front_index]:
            # Iterate through the individuals dominated by i
            for j in dominating_individuals[i]:
                # Decrement the dominated count of j
                dominated_count[j] -= 1
                # If j is not dominated by any other individual, add it to the next front
                if dominated_count[j] == 0:
                    next_front.append(j)
        # Move to the next front
        front_index += 1
        # Add the next front to the list of fronts
        if next_front:
            fronts.append(next_front)
    # Convert indices to individuals in the fronts
    fronts = [[evaluated_population[i] for i in front] for front in fronts]
```



```python
    return fronts

# Function to check dominance
def dominates(individual1, individual2):
    # Check if individual1 dominates individual2
    return all(i1 <= i2 for i1, i2 in zip(individual1, individual2)) and any(i1 < i2 for i1, i2 in zip(individual1, individual2))

def calculate_crowding_distance(fronts):
    crowding_distances = {}
    for front in fronts:
        front_size = len(front)
        # Initialize crowding distances for individuals in the front
        crowding_distances.update({individual: 0 for individual in front})
        # Calculate crowding distance for each objective
        for objective_index in range(len(front[0])):
            # Sort individuals in the front based on the objective value
            front.sort(key=lambda x: x[objective_index])
            # Set the crowding distance of the boundary individuals to infinity
            crowding_distances[front[0]] = math.inf
            crowding_distances[front[-1]] = math.inf
            # Calculate crowding distance for the inner individuals
            objective_range = front[-1][objective_index] - front[0][objective_index]
            if objective_range != 0:  # Check for divide by zero
                for i, ind in enumerate(front[1:-1], start=1):
                    crowding_distances[ind] += (front[i + 1][objective_index] -
                    front[i - 1][objective_index]) / objective_range
    return crowding_distances
```



```python
def select_population(fronts, crowding_distances, pop_size):
    selected_population = []
    current_size = 0
    front_index = 0
    # Select individuals from non-dominated fronts until the population size is reached
    while current_size < pop_size and front_index < len(fronts):
        front = fronts[front_index]
        # Sort individuals in the front based on crowding distance
        front.sort(key=lambda x: crowding_distances[x], reverse=True)
        # Add individuals from the front to the selected population
        for individual in front:
            if current_size < pop_size:
                selected_population.append(individual)
                current_size += 1
            else:
                break
        front_index += 1
    return selected_population

def crossover_and_mutation(selected_population, num_variables, mutation_rate):
    offspring_population = []
    while len(offspring_population) < len(selected_population):
        # Select two parents randomly from the selected population
        parent1 = random.choice(selected_population)
        parent2 = random.choice(selected_population)
        # Perform crossover to generate offspring
        offspring = crossover(parent1, parent2, num_variables)
        # Perform mutation on the offspring
        offspring = mutation(offspring, mutation_rate)
        # Add the offspring to the offspring population
        offspring_population.append(offspring)
    return offspring_population

def crossover(parent1, parent2, num_variables):
    # Perform crossover operation (e.g., one-point crossover)
    crossover_point = random.randint(1, num_variables - 1)
    offspring = parent1[:crossover_point] + parent2[crossover_point:]
    return offspring
```



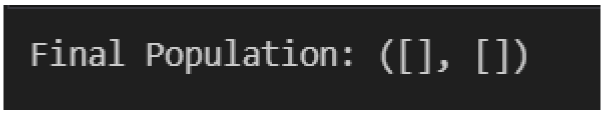
```python
def mutation(offspring, mutation_rate):
    # Perform mutation operation (e.g., bit-flip mutation)
    mutated_offspring = []
    for gene in offspring:
        if random.random() < mutation_rate:
            # Mutate the gene
            mutated_offspring.append(random.uniform(0, 10))
        else:
            mutated_offspring.append(gene)
    return mutated_offspring
```

```
0.0s
```

```python
pop_size = 20
max_gen = 100
num_variables = 7  # Specify the number of variables/dimensions
final_population = nsga2(pop_size, max_gen, num_variables)
print("Final Population:", final_population)
```

4) Results

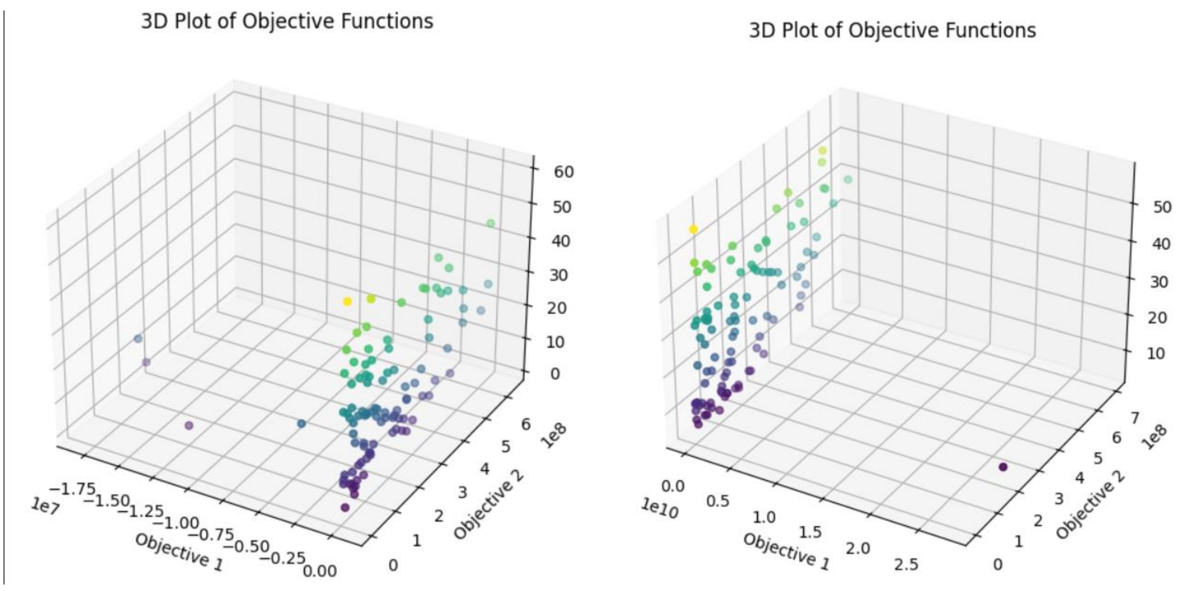
Via this classical NSGA code I got the empty results as the function generated the population randomly and then the to satisfy the constraints it had to reject the generated population individuals.



```
Final Population: ([], [])
```

Thus, I modified the code ensured that the during the time of the generation of the individuals I get those only which satisfy the and then the problem occurred that that code took hours but it kept on running without giving any result or the error.

Then I used the nsga along with the optimize library from the pymoo and made some changes in the code like mining the all objectives after implementing the negative sign on those which were to maximize then I got these results.



Still unsatisfied with the results I tried implementing the Particle Swarm Optimization (PSO) algorithm.

- PSO is a population-based optimization technique inspired by the social behaviour of bird flocking or fish schooling. In PSO, a population of candidate solutions, called particles, moves around the search space to find the optimal solution.

Particle Initialization: Each particle is initialized with a random position and velocity in the search space. The position represents a potential solution to the optimization problem.

Fitness Evaluation: The fitness of each particle is evaluated based on its position. The fitness function is defined by evaluate_fitness, which calculates the sum of the objective values for a given particle's position. If the particle violates any constraints defined by constraints, it is assigned a fitness of infinity to penalize infeasible solutions.

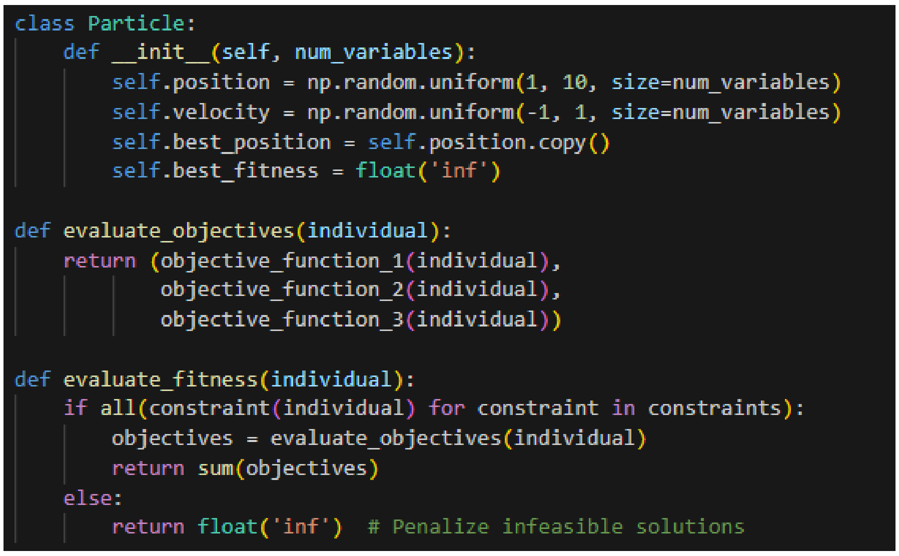
Updating Particle's Best Position and Global Best Position: Each particle remembers its best position (local best) and the corresponding fitness encountered so far. Additionally, the global best position and its fitness are tracked among all particles.

Velocity and Position Update: The velocity and position of each particle are updated using the current velocity, best position found by the particle itself (cognitive component), and the global best position found by any particle (social component). The update equations use inertia weight (w), cognitive weight (c1), and social weight (c2) parameters to control the influence of each component.

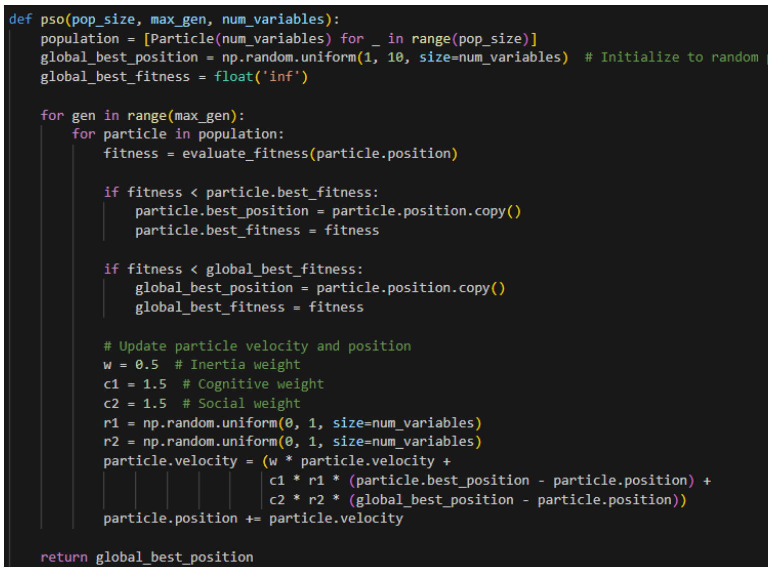
The update equations use parameters like inertia weight (w), cognitive weight (c1), and social weight (c2) to control the influence of each component.
The velocity update equation combines the particle's current velocity with two components:
The cognitive component, which is the difference between the particle's best position and its current position, scaled by c1.
The social component, which is the difference between the global best position and the particle's current position, scaled by c2.
These components guide the particles towards promising areas of the search space while allowing for exploration and exploitation.

Iterative Optimization: Steps 2-4 are repeated for a fixed number of iterations (max_gen) or until a termination condition is met.

Termination: After the specified number of iterations, the algorithm returns the global best position found, which represents the optimized solution.

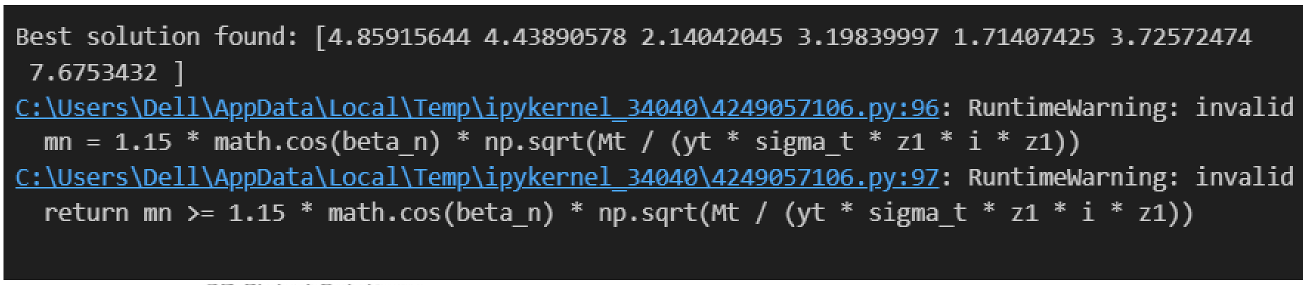


```python
class Particle:
    def __init__(self, num_variables):
        self.position = np.random.uniform(1, 10, size=num_variables)
        self.velocity = np.random.uniform(-1, 1, size=num_variables)
        self.best_position = self.position.copy()
        self.best_fitness = float('inf')

def evaluate_objectives(individual):
    return (objective_function_1(individual),
            objective_function_2(individual),
            objective_function_3(individual))

def evaluate_fitness(individual):
    if all(constraint(individual) for constraint in constraints):
        objectives = evaluate_objectives(individual)
        return sum(objectives)
    else:
        return float('inf')  # Penalize infeasible solutions
```

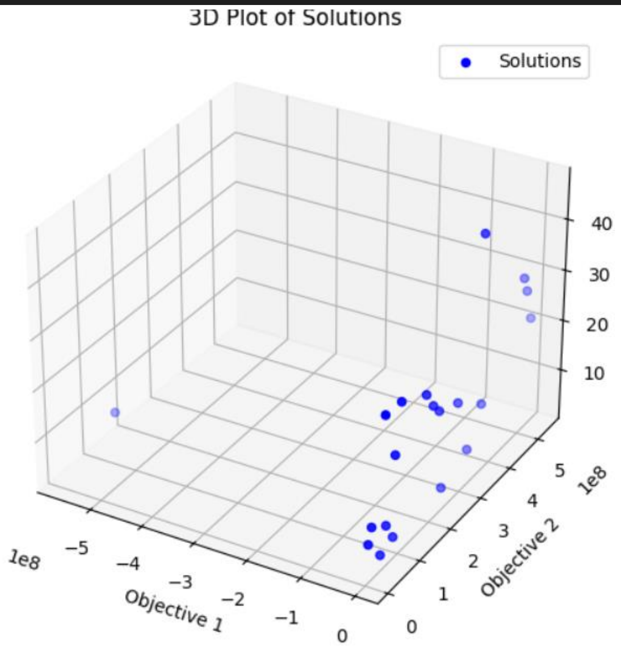


```python
def pso(pop_size, max_gen, num_variables):
    population = [Particle(num_variables) for _ in range(pop_size)]
    global_best_position = np.random.uniform(1, 10, size=num_variables)  # Initialize to random
    global_best_fitness = float('inf')

    for gen in range(max_gen):
        for particle in population:
            fitness = evaluate_fitness(particle.position)

            if fitness < particle.best_fitness:
                particle.best_position = particle.position.copy()
                particle.best_fitness = fitness

            if fitness < global_best_fitness:
                global_best_position = particle.position.copy()
                global_best_fitness = fitness

            # Update particle velocity and position
            w = 0.5  # Inertia weight
            c1 = 1.5  # Cognitive weight
            c2 = 1.5  # Social weight
            r1 = np.random.uniform(0, 1, size=num_variables)
            r2 = np.random.uniform(0, 1, size=num_variables)
            particle.velocity = (w * particle.velocity +
                                 c1 * r1 * (particle.best_position - particle.position) +
                                 c2 * r2 * (global_best_position - particle.position))
            particle.position += particle.velocity

    return global_best_position
```

Solution Obtained:



```
Best solution found: [4.85915644 4.43890578 2.14042045 3.19839997 1.71407425 3.72572474
 7.6753432 ]
C:\Users\Dell\AppData\Local\Temp\ipykernel_34040\4249057106.py:96: RuntimeWarning: invalid
  mn = 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))
C:\Users\Dell\AppData\Local\Temp\ipykernel_34040\4249057106.py:97: RuntimeWarning: invalid
  return mn >= 1.15 * math.cos(beta_n) * np.sqrt(Mt / (yt * sigma_t * z1 * i * z1))
```

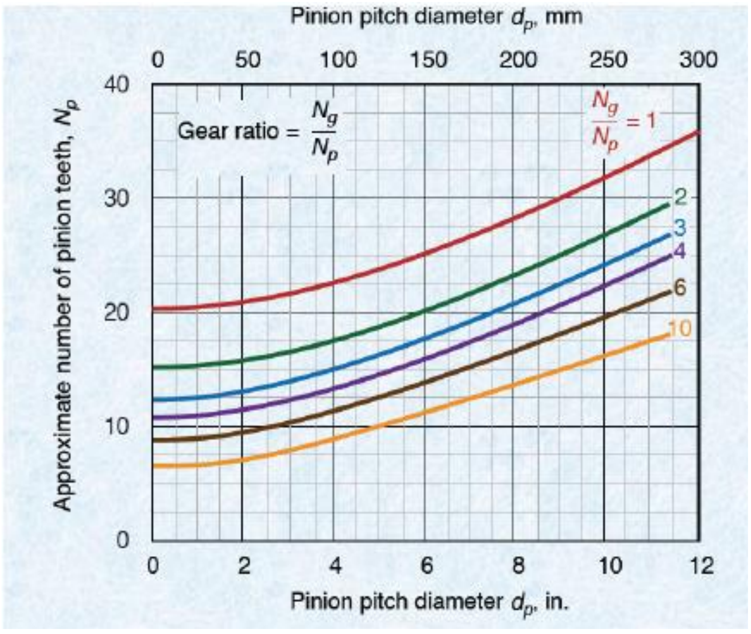


This algorithm aims to converge towards the optimal solution by adjusting the velocity and position of particles based on their individual experiences and the global best found so far.

5) Discussion and Conclusion

Spiral bevel pair optimization involves a large number of intricate objective functions and constraints, and it takes a long time to compute. The comparison demonstrates that by maximizing the three objective functions without violating any constraints, the results produced by the NSGA-II software are superior. With the least amount of processing time and effort, the results were acquired.

Accompanying the gain in efficiency was decrease in weight.



Three goal functions are taken into consideration in order to enhance the performance of bevel gears utilized in diverse industrial applications: minimizing weight, maximizing transmission efficiency, and minimizing cone distance.

We can utilize data from the current bevel gear design, which is employed in industries, to compare the code's outputs and results. It turns out that this design is superior in terms of cost, efficiency, and gear compactness.

6) References

https://www.sciencedirect.com/science/article/pii/S0094114X21002561
https://github.com/sahutkarsh/NSGA-II/blob/master/NSGA-II.ipynb
https://pymoo.org/algorithms/moo/nsga2.html

7) Contribution

- Ayush Kumar Singh (22b2203) :- Coded the whole NSGA part and tried different algorithms and models. Made the Presentation and the whole Report.
- Shahu Patil (22b2146) :- Decided that NSGA algorithm will be applied and cited the references for the documentation.
- Rithvik Subash (22b2120) :- Researched on the reference papers and theoretical formulae given for the objective functions.