Parameter initialization

```
clear;
p.nGenerations = 100;
p.nPopulation = 30;
p.nOffspring = 30;
p.nGenes = 15;
p.crossoverProbability = 0.9;
p.mutationProbability = 3 / p.nGenes;
```

Initialize population

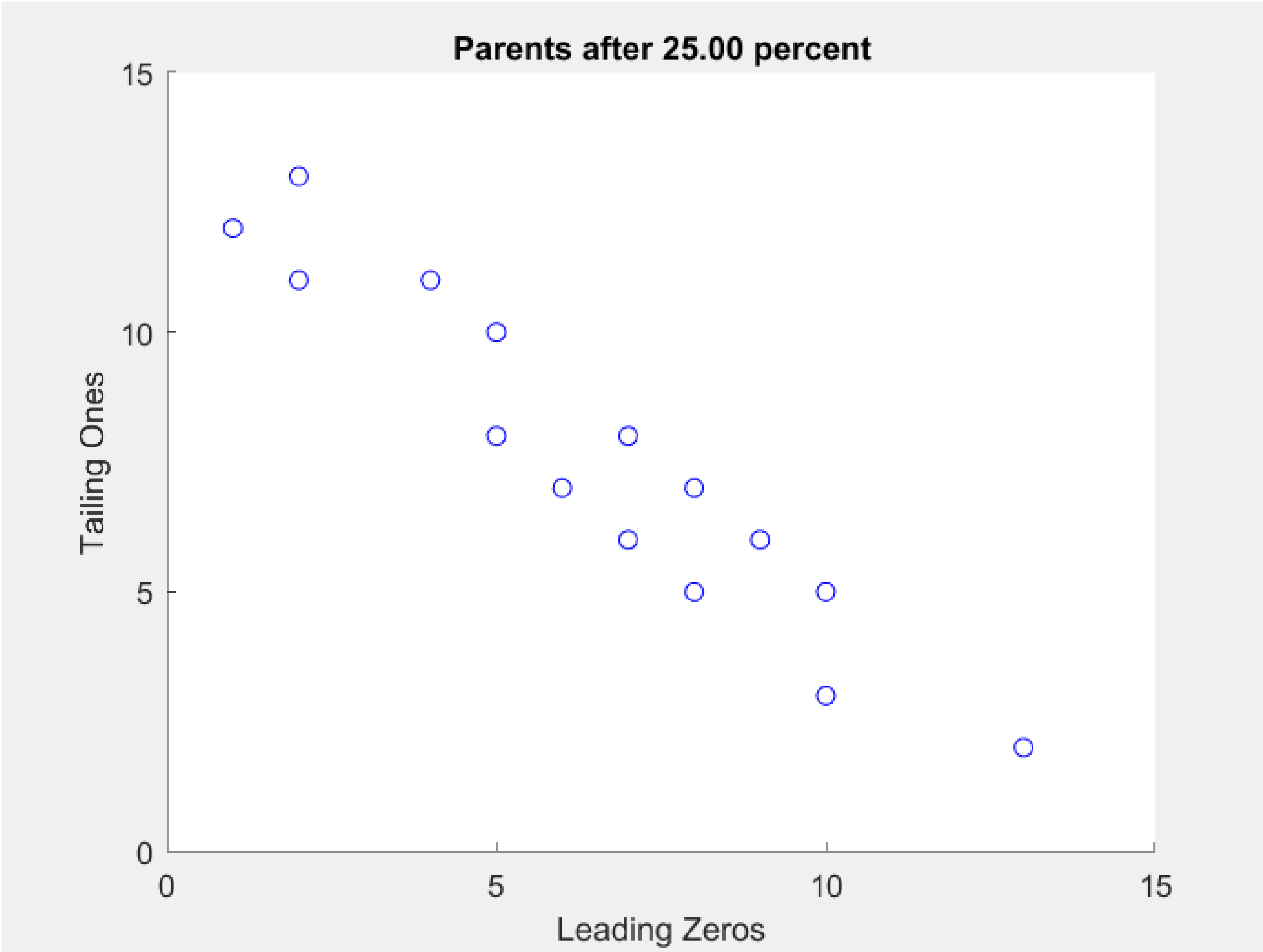
```
p.population = randi([0 1], p.nPopulation, p.nGenes);
```

Evolution Loop

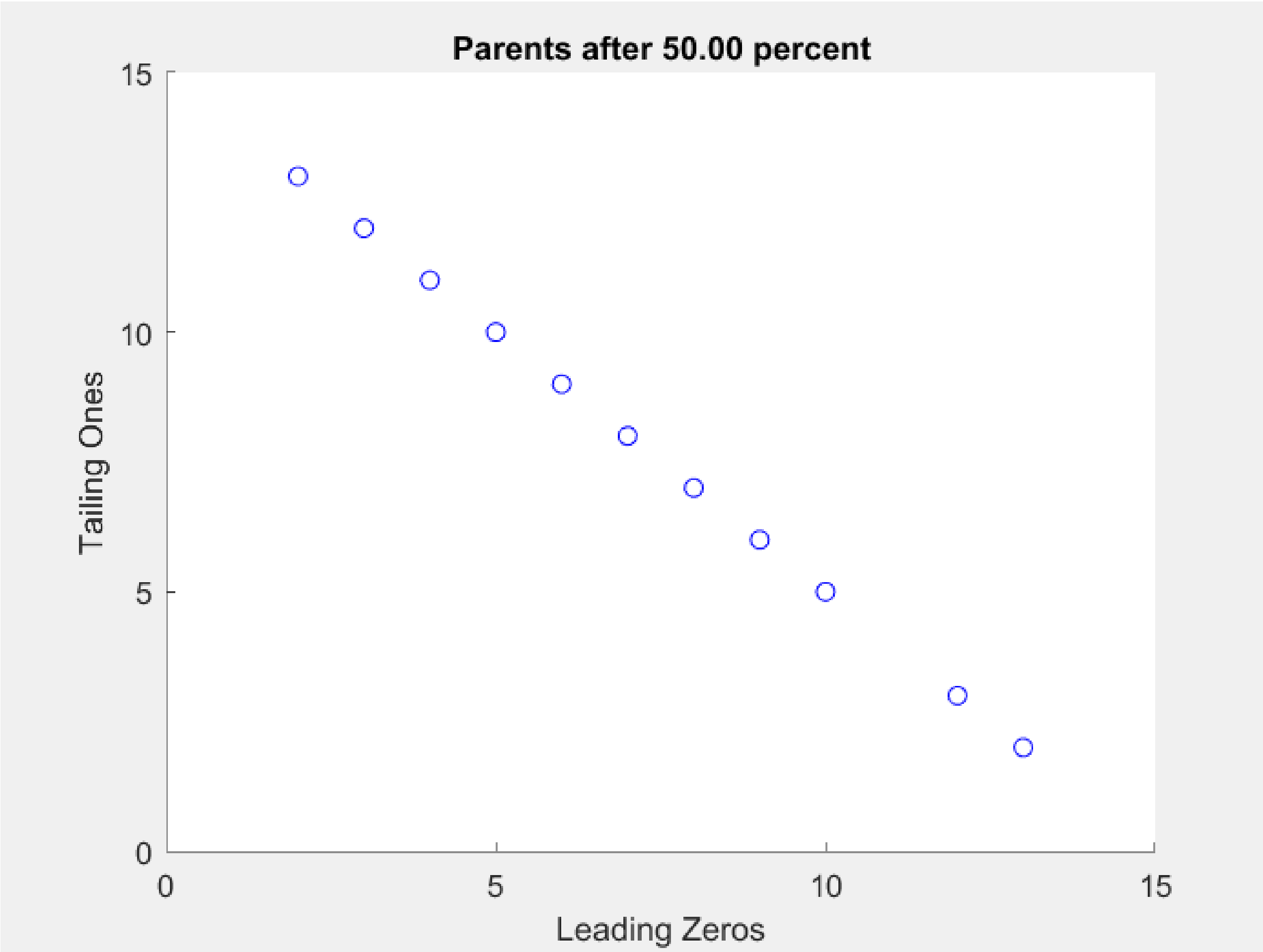
```
visualizationStep = 1;
for generation=1:p.nGenerations

    % Evaluation
    %   Computes the number of leading zeros and tailing ones for every
    %   individual within the population and stores both values as fields
    %   in the result.
    fitness = computeFitness(p.population);
    p.leadingZeros = fitness.leadingZeros;
    p.tailingOnes = fitness.tailingOnes;

    % Visualization
    %   Plots the parents every generation and at 25, 50. 75 and 100
    %   percent of maximum generations.
    if generation >= p.nGenerations*(0.25 * visualizationStep)
        visualizationStep = visualizationStep + 1;
        figure(visualizationStep);clf;hold on;
        title(sprintf('Parents after %.2f percent',...
            (visualizationStep - 1) * 25));
        plot(p.leadingZeros, p.tailingOnes,'bo');
        xlabel('Leading Zeros');
        ylabel('Tailing Ones');
        axis([0,p.nGenes,0,p.nGenes]);
    end

    figure(1);clf;hold on;
    subplot(1,2,1);
    plot(p.leadingZeros, p.tailingOnes,'bo');
    title(sprintf('Parents at generation %d', generation));
    xlabel('Leading Zeros');
    ylabel('Tailing Ones');
    axis([0,p.nGenes,0,p.nGenes]);

    % Create and evaluate offspring
    %   Generates offspring for the given population by using tournament
    %   selection, mutation and crossover as has been used in oneMax.
    %   Elitism is not used because the pareto fronts takes care of the
    %   best individuals.
    offspring = generateOffspring(p);
```

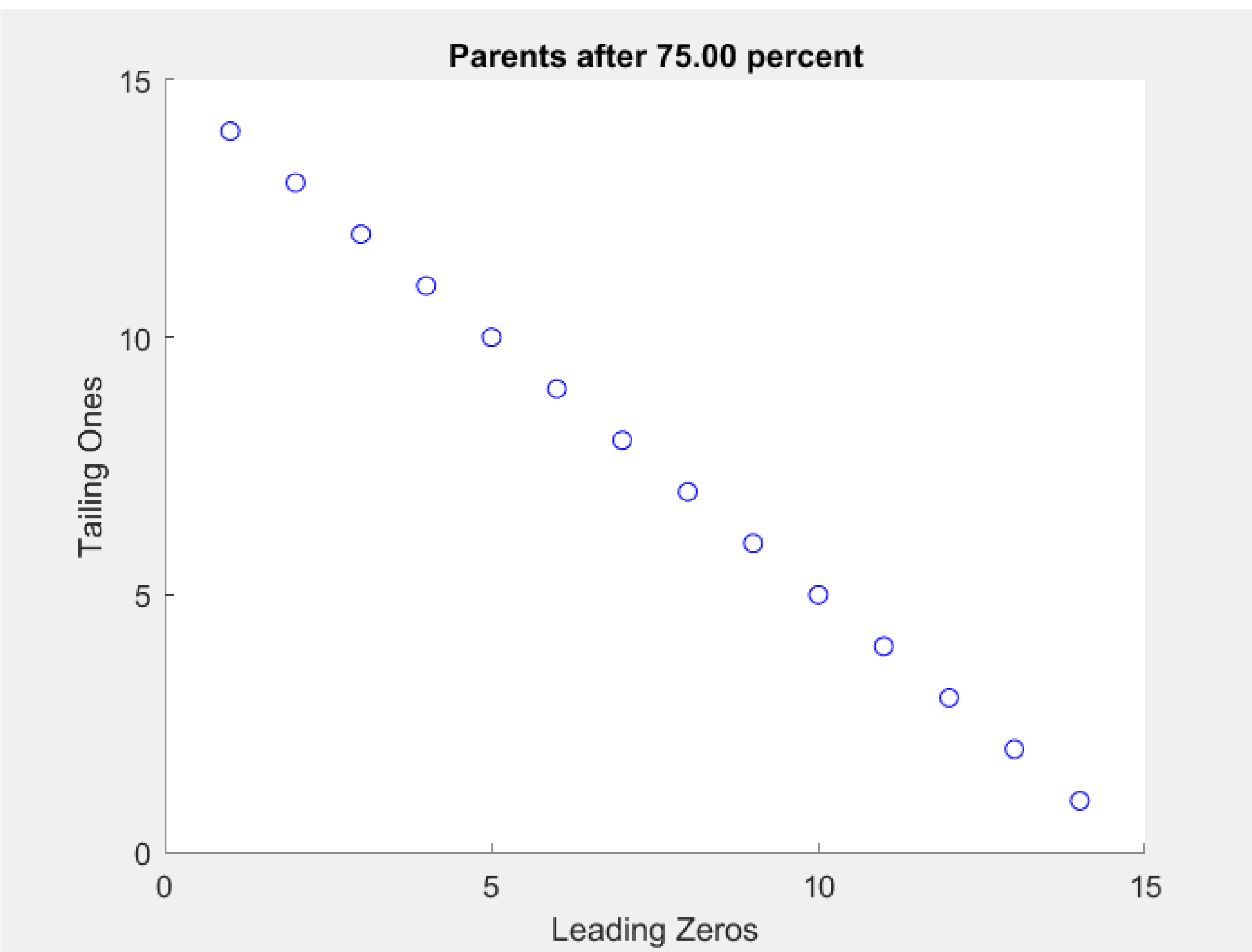
```
    fitness = computeFitness(offspring);
    p.leadingZeros = [p.leadingZeros fitness.leadingZeros];
    p.tailingOnes = [p.tailingOnes fitness.tailingOnes];
    p.population = [p.population;offspring];

    %   Computes the domination sets and counts for every individual and
    %   assigns each individual to the pareto front it belongs to.
    %   This is done by checking for each individual those individuals it
    %   dominates and is dominated by. If there is no dominating
    %   individual, the individual goes into the front.
    %   For every next front the previous fronts get subtracted from each
    %   remaining individual's domination counter. Again, if one of the
    %   remaining individual's counter decreases to zero, it belongs to the
    %   next front.
    %   More details can be found in dominationSort.m
    paretoFronts = dominationSort(p);

    % Choose individuals for next generation using pareto fronts.
    %   As long as the next front fits into next generation, add...
    nextPopulation = [];
    i = 1;
    while size(nextPopulation,1) + length(paretoFronts{i}) <=...
            p.nPopulation
        indizes = paretoFronts{i};
        paretoElements = p.population(indizes,:);
        nextPopulation = [nextPopulation; paretoElements];
        i = i + 1;
    end

    %   If next generation is not full yet,...
    if size(nextPopulation,1) ~= p.nPopulation
        %   ...fill it with best spreaded individuals from next front.

        %   Given a population and its values for leadingZeros and
        %   tailingOnes, this function computes the crowding distance for
        %   every individual.
        distances = computeCrowdingDistance(p.population(paretoFronts{i},:));

        %   Sort individuals according to descending distance...
        [sortedDistances, iSortedDistances] = sort(distances,'descend');

        %   ... and carry over the individuals with the highest values.
        sortedPareto = paretoFronts{i}(iSortedDistances);
        nIndividualsLeft = p.nPopulation - size(nextPopulation,1);
        bestSpacedIndividuals = p.population(...
            sortedPareto(1:nIndividualsLeft),:);
        nextPopulation = [nextPopulation;bestSpacedIndividuals];
    end

    % Visualization
    subplot(1,2,2);
    plot(p.leadingZeros, p.tailingOnes,'bo');
    title(sprintf('Whole Generation %d', generation));
    xlabel('Leading Zeros');
    ylabel('Tailing Ones');
    axis([0,p.nGenes,0,p.nGenes]);
    pause(0.1);

    p.population = nextPopulation;
```

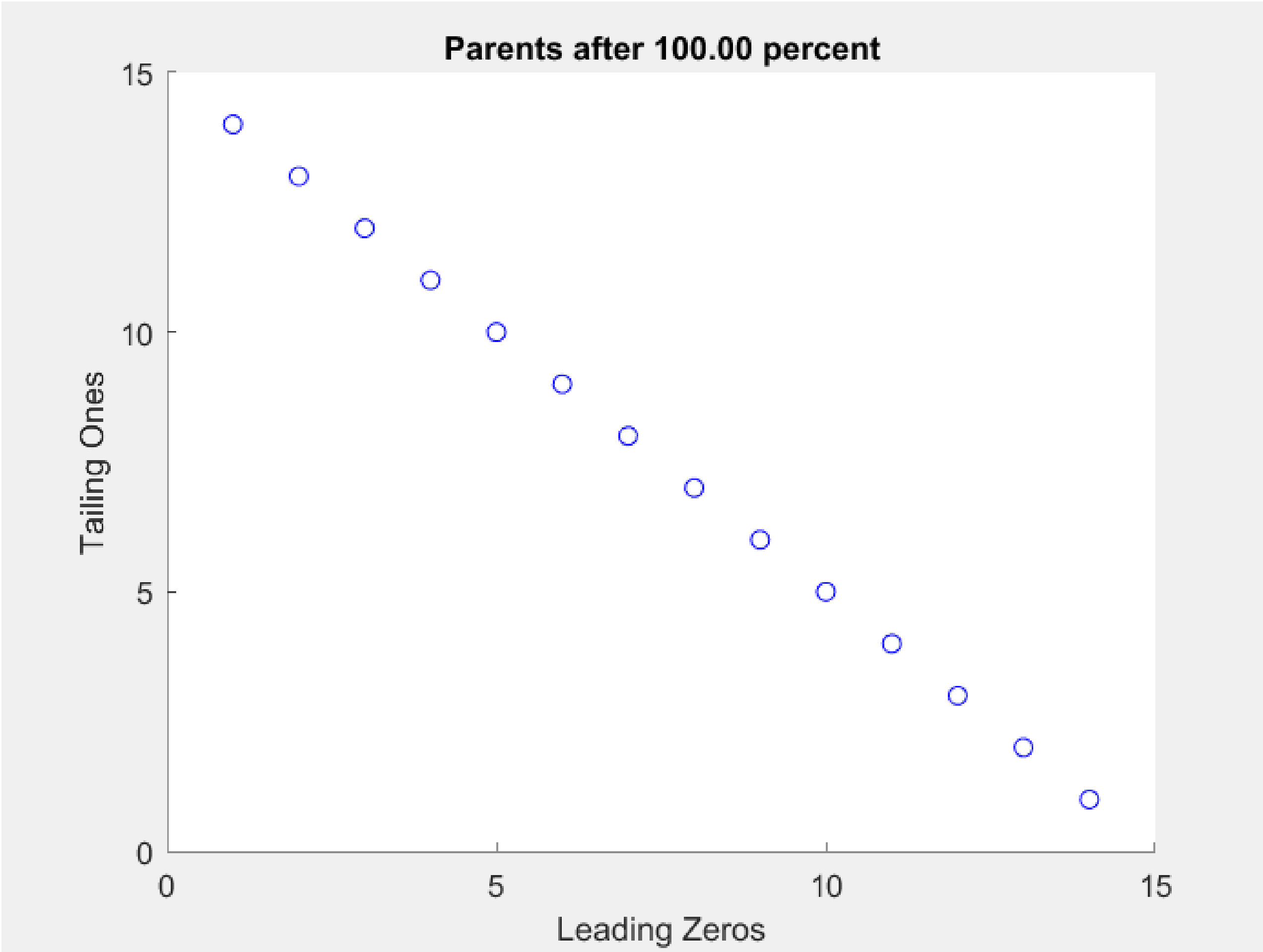
```
end
```

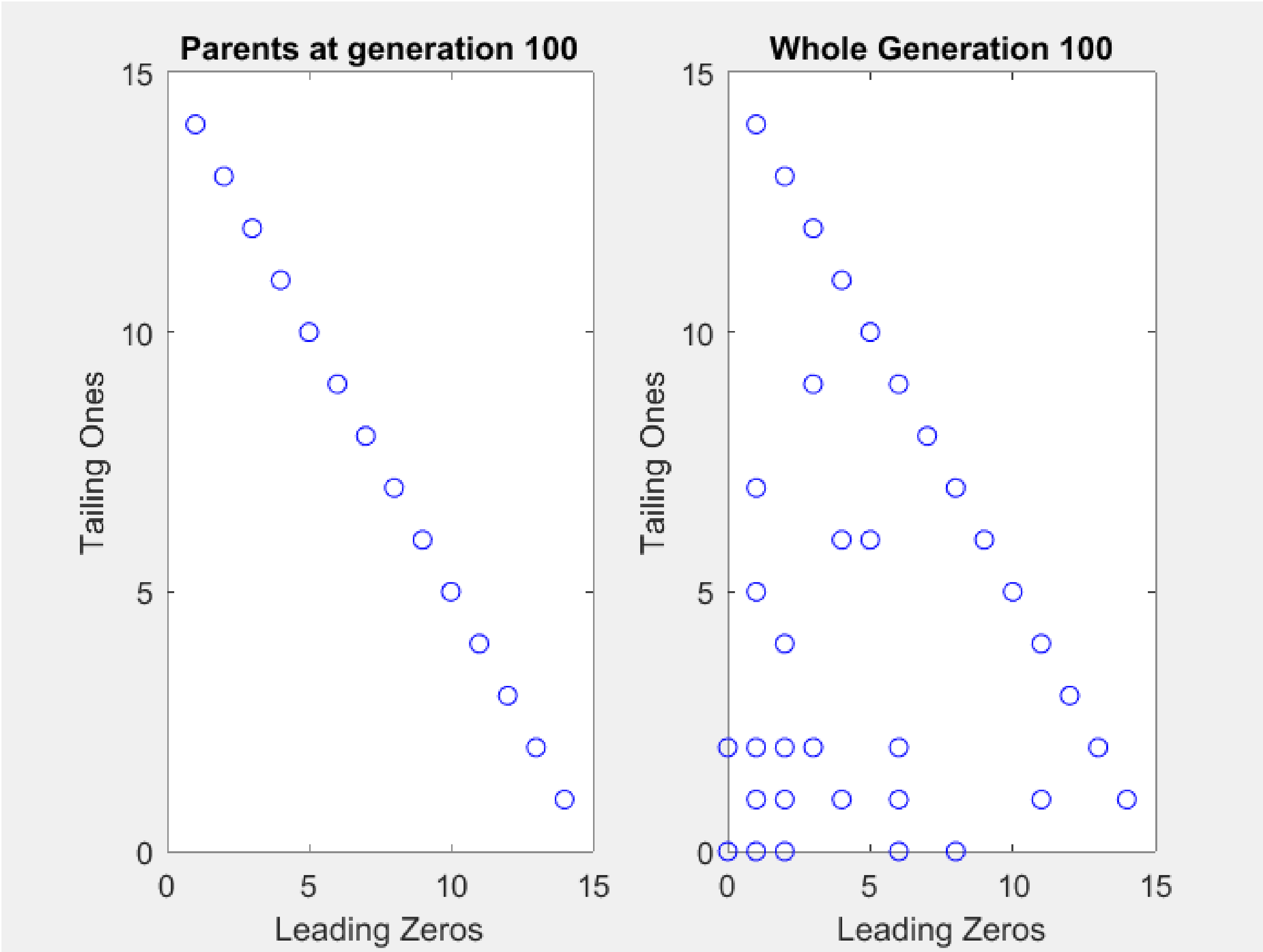


Parents after 25.00 percent
Tailing Ones
Leading Zeros
0
5
10
15


Parents after 50.00 percent
Tailing Ones
Leading Zeros
0
5
10
15
0
5
10
15


Parents after 75.00 percent
Tailing Ones
Leading Zeros
0
5
10
15
0
5
10
15


Parents after 100.00 percent
Tailing Ones
Leading Zeros
0
5
10
15
0
5
10
15




```
population = p.population
```

```
population = 30x15 double

     0     0     0     0     0     0     0     0     0     0     0     0     0 ···
     0     1     1     1     1     1     1     1     1     1     1     1     1
     0     1     1     1     1     1     1     1     1     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     1     1
     0     0     1     1     1     1     1     1     1     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     0     0     0     0     0     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     1     1     1     1     1     1     1     1     1     1
     ⋮
```

Comment on results

```
% It seems as if NSGA-II has some problems in extrapolating. It only rarely
% comes up with all ones or all zeros solutions.
```