AE4879 Mission Geometry and Orbit Design

Assignment 6: Optimization II
OPTIM-7

Simon Billemont

October 15, 2010

Assignment 6: Optimization II

This assignment deals with a test case for a Generic Algorithm optimizer. The considered optimization problem is the maximalization of the Himmelblau function over the interval of 0 to 5 for both parameters.

1. Generic Algorithm

A Genetic Algorithm optimizer (GA) is an optimization technique that resembles the way life evolved on this planet. It basically starts with a pool or generation of purely random data, that evolves or changes in such a way that the end result is in a more optimal then the initial state. How good that the generation is, can be determined by the optimization function. The evolution of a generation (making a new generation) consists of the following major phases: Reproduction, Carryover, Immigration and Mutation.

1.1. Reproduction and Carryover

In these phases, the current generation is used to lay the foundations of the next generation. In the Reproduction phase, the current generation is combined so that a set of children for the next generation are formed. Then in the Carryover phase, the most optimal parents are also transfered to the next generation. This ensures that the best qualities of a generation are passed on.

The reproduction is done by first selecting suitable parents to match. This is done by randomly inserting parents into a list. Optionally, a weighted selection can be done. Then in sequence, the list is iterated and two subsequent parents produce at least one child.

The creation of a child is done by combining the data of the two parents. This data can be represented in several ways, but in this document it is assumed to be a binary sequence. In this sequence, a predefined set of bits form the first parameter and the remaining bits the second parameter of the Himmelblau function. Equivalently to the binary sequence, an equivalent integer can be used. This simplifies the representation, and is simpler for computers to handle. The combination of two parents is done by making a random bit mask, and giving (based on the mask) bits from parent one to the child. Then the bits of parent two are transfered using the binary complement of the mask. Mathematically this process can be expressed as in eq 2.

$$p_1, p_2, \text{mask} \, \epsilon \, [0, 2^{\text{length}} - 1] \tag{1}$$

$$c = (p_1 \wedge \text{mask}) \vee \left(p_2 \wedge \overline{\text{mask}}\right) \tag{2}$$

Where p_1, p_2 are the parents and c is the resulting child. length Is the total number of bits per parent and mask is the binary mask used in the combination of the parents. Note that the AND ($\wedge$) and the OR ($\vee$) operations are done on the binary string.

1.2. Immigration

Immigration is the phase where new, random, instances are created (see eq 3). The consequence of these is that the entire search space of the optimization problem is considered. Thus looking for alternative optima in the functional domain. When an immigrant is more optimal, it is automatically pushed to the next generation in the Carryover phase.

$$i \, \epsilon \, [0, 2^{\text{length}} - 1] \tag{3}$$

1.3. Mutation

In the mutation phase, the generation is exposed to random changes in the population data. This means that random bit flips in some instances happen. In order to preserve the current optimum, the most optimal top of the generation is shielded from these effects. The mutation process injects alternatives into the current generation. This is done, because otherwise do to the reproduction phase, certain solutions are gradually being erased (eg a solution that is close to a very bad location).

2. Optimization problem

- **Develop a general Genetic Algorithm program**
- **Apply this to maximize the Himmelblau function (problem Bbis, see [1]). Select the length of the bit representation such that the resolution is 32 grid points for each parameter. Make parent selection fully random (i.e., even chances). Ignore immigration. Set mutation chances at 0.1%. Repeat until convergence at 0.1% , or realistic maximum number of generations achieved. Repeat each computation 3 times.**
- **Idem, for a parameter resolution of 128 grid points.**
- **Idem, for a parameter resolution of 512 grid points.**
- **Discuss the results of the previous steps (parameter values, function value, computation time) and compare to the true optimum**

The problem considered for the rest of the document is the Himmelblau function defined by eq 4 (see [1]). The optimization problem considered is to minimize the function over the domain in eq 5.

$$f(x_1, x_2) = \left(x_1^2 + x_2 - 11\right)^2 + \left(x_1 + x_2^2 - 7\right)^2 \tag{4}$$

$$\mathbb{D} : x_1 \epsilon [0, 5] \text{ and } x_2 \epsilon [0, 5] \tag{5}$$

The GA algorithm creates a population of 100 immigrants. This is the initial population. Then it goes through all the phases of the GA algorithm, where each set op parents creates 2 children to maintain a constant population. For the mutation sequence, 10% of all the mutatable instances mutates. This is iterated untill 10 subsequent generations

	Carryover	Bits / param	Grid size	x_1	x_2	$f(x_1, x_2)$	time [s]	generations
Run 1	0 %	5	32x32	5.00	5.00	890.00	4.70	500
Run 2	0 %	5	32x32	4.84	3.71	394.51	4.77	500
Run 3	0 %	5	32x32	4.68	4.19	460.13	4.78	500
Run 4	1 %	5	32x32	5.00	5.00	890.00	0.25	22
Run 5	1 %	5	32x32	5.00	5.00	890.00	0.19	18
Run 6	1 %	5	32x32	5.00	5.00	890.00	0.16	17
Run 7	0 %	7	128x128	4.37	4.88	618.03	4.58	500
Run 8	0 %	7	128x128	4.21	4.88	578.12	4.55	500
Run 9	0 %	7	128x128	4.72	4.80	692.38	4.65	500
Run 10	1 %	7	128x128	5.00	4.84	815.14	0.13	11
Run 11	1 %	7	128x128	5.00	5.00	890.00	0.29	30
Run 12	1 %	7	128x128	4.84	5.00	890.00	0.20	21
Run 13	0 %	8	256x256	1.69	4.96	382.53	4.62	500
Run 14	0 %	8	256x256	4.65	4.76	650.03	4.60	500
Run 15	0 %	8	256x256	4.98	3.71	443.82	4.69	500
Run 16	1 %	8	256x256	5.00	5.00	890.00	0.28	26
Run 17	1 %	8	256x256	4.96	4.96	854.26	0.17	18
Run 18	1 %	8	256x256	4.98	4.98	872.01	0.25	27

Table 1: Results of the GA algorithm on the Himmelblau function with different grid sizes

yield the same result (tolarence of 10^{-1}). A couple of subsequent runs where perfomed, and the details and performance has been tabulated in table 1. For each grid size, three runs of the GA algorithem where done. This is because due to the varius ranomizations in the algorithm. A graphical evolution of the Run 5 is also given in fig 1. It shows visually several generations, with the origin of each datapoint: immigrant, child, parent (carryover).

The most important observation is that without carryover, the algorithm takes a lot more generations to get somewhat of a result (the computations are stopped after 500 generations). With moving a single (best) parent into the following generation, we see that it has a profound effect on the convergence speed of the GA algorithm.

One also notices if the parameter precision is increased (more bits/param) then also GA takes more time or generations to find the optimal result. From table 1, one reads that with 5 bits/param, GA takes about 19 generations to find the result. For 7 bits/param this increases to 20 generations, and finally for 8bits/param the GA algorithm uses on average 23 generations. What this shows is that the parameter size only slightly affects the GA calculation time.

Furthermore, GA does not always give the optimum result in the search area. Note that for GA to stop, 10 subsequent generations have to yield the same result. This means that the new children and mutations have a small probability of improving the result. However the final values are very close to the optimum. This actual optimum can then be using other optimization techniques.

Concluding, one finds that genetic algorithms are a good and relatively fast way to find the region of the optimum of a given function. However for accurate results either large bit sequences are needed, with lots of generations making it slow to achieve the absolute optimum. For this alternative optimization techniques such as steepest gradient techniques can be used.

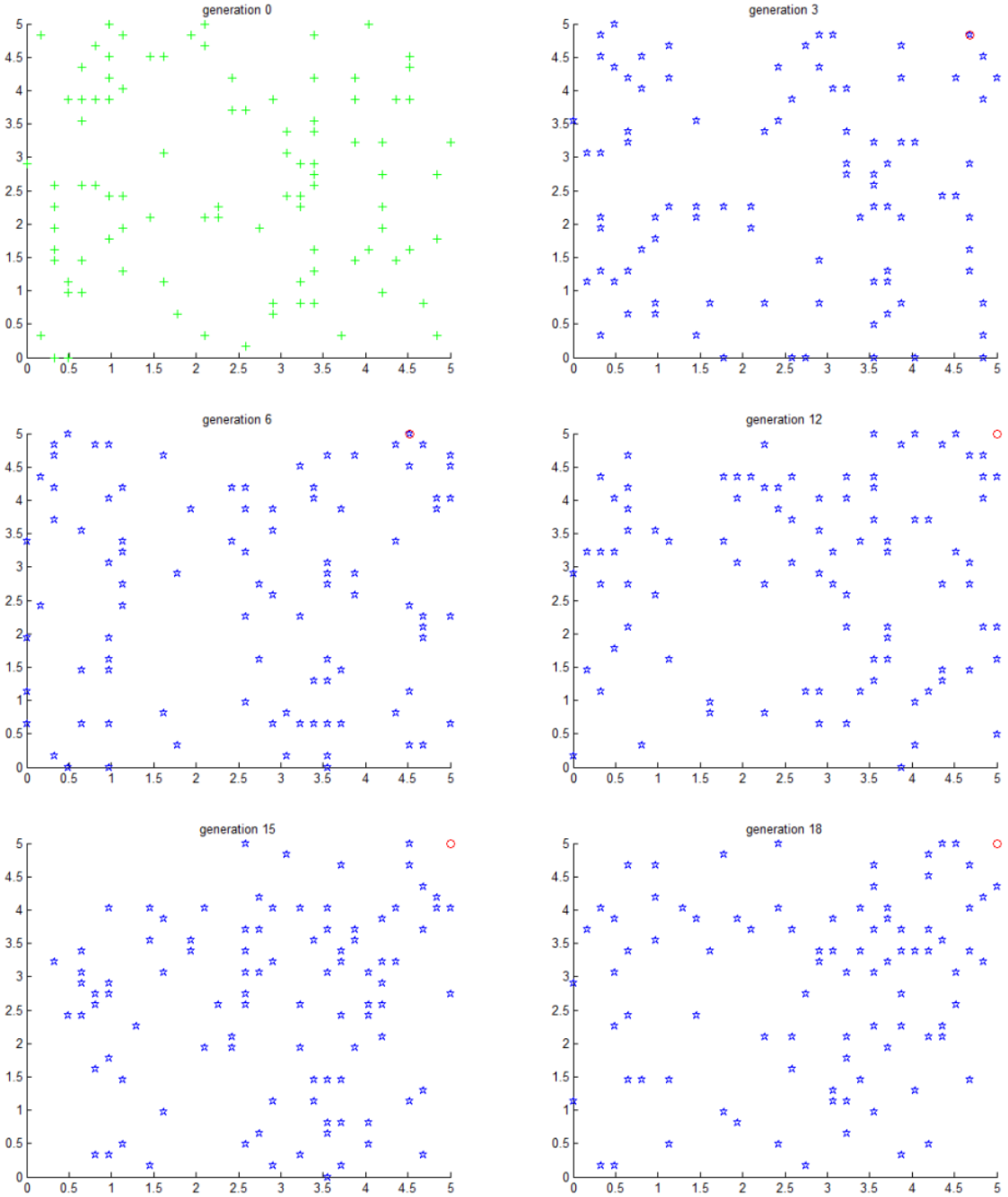

Figure 1: Graphical representation of different generations in Run 5. Marker style depicts the origin of each data point: immigrants (gree, $+$), children (blue, $\star$) and parents (red, $\circ$)

References

[1] R. Noomen, *AE4-879 Optimisation V3.2*, TUDelft Lecture Slides, 2010.

[2] MathWorks. (2010a) Matlab 7.11. Natick, MA.

[3] J. R. Wertz, *Orbit & Constellation Design & Management*, second printing ed. El Segundo, California: Microcosm Press, 2009.

[4] WolframResearch. (2008) Mathematica edition: Version 7.0. Champaign, Illinois.

Additional information

Estimated work time:

$\sim$ 2h Studying theory + $\sim$ 5h making assignment + $\sim$ 5h writing report = $\sim$ 12h

Made by

Simon Billemont

Stud Nr: 1387855

s.billemont@student.tudelft.nl

Version history

Version 1: Initial document

A. Matlab source code

The code written to implement the three described optimizers was written in MATLAB 7.11 (2010b)[2]. A structured overview of the dependencies is given below:

- OPTIM7.m
  - Himmelblau.m
  - FunctionBridge.m
  - Combine.m
  - GAoptimizer.m
    - Generation.m

The script OPTIM7.m does all the configures GAoptimizer to perform a specific optimization. For this, it wraps the Himmelblau function (Himmelblau.m) into a function that converts the bit sequence into x_1 and x_2 with corresponding $f(x_1, x_2)$. Then GAoptimizer creates generations (Generation.m). The combining of parents into children is done with the Combine function (Combine.m).

Listing 1: OPTIM7.m: Script to initialize the optimizer in a specific configuration

```
%% Setup environment
clc
clearvars
close all

% Make a new utility for saving pictures
saver = ImSav();
saver.c_plotsDir = '../images/matlab/';     % Where the plots will go
saver.cleanPlots;                           % Remove any plots already made
saver.c_changec_figSize = 0;                % Do not resize
saver.c_appendFigureNr = 0;                 % Do not append stuff the the name

%% Setup constants

x1Bits = 5;
x2Bits = x1Bits;

% Make the himmbelblau function handle
[func, fromParam] = FunctionBridge(@Himmelblau, x1Bits, x2Bits, [0,5], [0,5]);

% Basic GA initialization
opt = GAoptimizer(func, x1Bits+x2Bits);

%% Setup optimizer
% Convert the propagation to pure random (parent selection)
opt.c_propagationWeightFunc = @(x) rand(size(x));
% Function to combine the parents to children
opt.c_combineFunc =           @Combine;
% # of children / pair f-of parents
opt.c_children = 2;
% 1% of parents are moved to the next generation
opt.c_parentCarryover = 0.01;
% 10% mutation chance
opt.c_mutationProbability = 0.1;
% Everyone can mutate (0 of the best are protected)
opt.c_mutationSafe = 0;
% Iterate until 10 subsequent results are equal
opt.c_iterationEqual = 10;
% Consider the result the same with 0.1 tolarence
opt.c_iterationTolerance = 0.1;
% Perform a maximi-um of 500 iterations (500 + 11 with 11 being initial setup count)
opt.c_maxIter = 511;
% Maximum population of a generation
opt.c_generationSize = 100;
% Maximize the himmeblau (ascend for minimization)
opt.c_mode = 'descend';

%% Compute the results
[gen, evolution] = opt.optimize;
```

Listing 2: Himmelblau.m: Function representation of the Himmelblau

```
function [ y ] = Himmelblau( x1, x2 )
%HIMMELBLAU function
%   y = ( x1.^2 + x2 - 11 ).^2 + ( x1 + x2.^2 - 7 ).^2

    if (nargin == 1) % assume vector input instead of seperate numbers
        x2 = x1(:,2);
        x1 = x1(:,1);
    end

    y = ( x1.^2 + x2 - 11 ).^2 + ( x1 + x2.^2 - 7 ).^2;
end
```

Listing 3: FunctionBridge.m: Create a bridge for the GA to call the Himmelblau

```
function [ b, fP ] = FunctionBridge(func, x1Bits,  x2Bits, x1Range, x2Range)
% Convert a function to be callable with a bit sequence istead of x1 and x2
    function [value] = bridge(param)
        % This computes f(x1,x2) based of the bit sequence

        % Split the param number (a bitwise representation)
        % into the two seperate parameters
        [x1, x2] = fromParam(param);
        % Compute the actual functional value
        %himmelblau = Himmelblau(x1, x2)';
        value = func([x1, x2])';
    end
    function [x1, x2] = fromParam(param)
        % convert the bit sequence into two parameters
        % Binairy representation of the number
        bin = dec2base(param,2, x1Bits + x2Bits);
        % Convert the subset of bits to a decimal number: sum 2^bitOffset
        x1 = bin2dec(bin(:,1:x1Bits));
        x2 = bin2dec(bin(:,x1Bits+1:x1Bits+x2Bits));
        %% Map to range
        x1 = (x1/((2^x1Bits)-1))*(x1Range(2)-x1Range(1))+x1Range(1);
        x2 = (x2/((2^x2Bits)-1))*(x2Range(2)-x2Range(1))+x2Range(1);
    end

    % Make the function to compute the himmelblau from a set of bits
    b =  @bridge;
    fP = @fromParam;
end
```

Listing 4: Combine.m: How parents are combined into children

```
function [ children ] = Combine( parents, nrChildren )
%COMBINE Combine a maximum of two parents into a specified # children
%   function [ children ] = Combine( parents, nrChildren )

    clazz = class(parents(1));  % Like uint8
    % Make a specified amount of children
    children = zeros(nrChildren, 1, clazz);
    for i=1:nrChildren
        % Make a random bit mask (random number between 0 and the maximal value)
        mask = randi(intmax(clazz), clazz);
        % Take the ones in the mask from parent 1 and the zeros from parent 2
        children(i) = bitor(bitand(mask, parents(1)), bitand(bitcmp(mask), parents(end)));
    end
end
```

Listing 5: GAoptimizer.m: Generic Algorithm optimizer

```
classdef GAoptimizer < handle
    %GAOPTIMIZER Genertic algorithem
    %   Optimizer that creates a set of evolving generations to solve an
    %   optimization problem

    properties
        c_func                          % Optimization function
        c_funcParamBits                 % Bit sequence length
        c_iterationTolerance = 1E-3;    % Tolarence to consider equal results
        c_iterationEqual =     5;       % end criteria (stop if 5 equal subsequent results)
        c_mode = 'ascend';              % maximize or minimize
        % 'ascend'  = minimize
        % 'descend' = maximize

        c_propagationWeightFunc = @(x) x; % Default is unweighted, how parents are selected
        c_mutationWeightFunc =    @(x) x; % Default is unweighted, how mutations are selected
        c_combineFunc =           @Combine;% Function to combine two parents to a child

        c_mutationProbability = 1E-3;     % Chances of mutation
        c_mutationSafe =        15;       % Protect from mutation, 15 best
        c_parentCarryover =     0.1;      % Amount of parents going to the newt gen in %
        c_generationSize =      50;       % Maximum population
```

```
        c_children =            1;        % Per pair make # childeren
        c_maxIter =             1E3;      % Force stop after # iterations
        currentGeneration %used to keep track of the current generation
    end

    methods
        % Constructor
        function obj = GAoptimizer(func, funcParamBits)
            obj.c_func          = func;
            obj.c_funcParamBits = funcParamBits;
        end
        function [gen, evolution] = optimize(obj)
            % initial population
            gen = obj.initialize;
            % initialize lists to store the generations
            obj.currentGeneration.sort;
            evolution = [inf*ones(1,obj.c_iterationEqual), obj.currentGeneration.paramVals(1)];
            i = obj.c_iterationEqual+1;
            % Find optimum
            while (~obj.checkEnd(evolution, i) && i < obj.c_maxIter)
                % Make the next generation
                gen = [obj.propagate, gen];
                % Sort the generation, so the best ones are at index 1
                obj.currentGeneration.sort;
                % Find the best instance of the generation and store the result
                evolution(end+1) = obj.currentGeneration.paramVals(1);
                i = i+1;
            end

        end
    end
    methods(Access=private)
        function e = checkEnd(obj, evol, i)
            e = 1;
            % 1 if the last c_iterationEqual numbers are equal
            for offset=1:obj.c_iterationEqual
                e = e && (abs(evol(i)-evol(i-offset)) < obj.c_iterationTolerance);
            end
        end
        function gen = initialize(obj)
            % Create an empty generation
            gen = Generation(obj.c_func, obj.c_funcParamBits, obj.c_mode);
            % Populate it with random instance
            gen.migrateInto(obj.c_generationSize);
            obj.currentGeneration = gen;
        end
        function newGen = propagate(obj)
            % shorthand
            gen = obj.currentGeneration;

            % Make a new generation of childeren based on the current generation
            newGen = gen.propagate(obj.c_propagationWeightFunc, obj.c_combineFunc, obj.c_children);

            % Sort the old generation on best parameters first
            gen.sort;
            % Select the best parents
            parents = gen.params(1:round(length(gen.params)*obj.c_parentCarryover));
            % Copy them into the new generation
            newGen.inject(parents, 1);

            % Fill the remaining slots with new (random) parameters
            newGen.migrateInto(obj.c_generationSize-length(newGen.params));

            % Mutate the population
            newGen.mutatePopulation(obj.c_mutationProbability, ...
                            obj.c_mutationSafe, obj.c_mutationWeightFunc);

            % Make shure we dont exeed c_generationSize
            newGen.sortParam
            newGen.limitPop(obj.c_generationSize);

            % Set current generation to the new developed generation
            obj.currentGeneration = newGen;
        end
    end

end
```

Listing 6: Generation.m: Holds data and can manipulate the generaion

```
classdef Generation < handle
    %GENERATION Hold all the data of a single GA generation

    properties
        c_func                      % Optimization function
        c_funcParamBits             % Bit sequence length
        c_mode = 'ascend';          % maximize or minimize

        % Stores the bit sequences values
        params
        % Stores the functional value for each bit sequence
        paramVals
```

```
        % Keeps track of where the nodes come from (unknown=0, parent=1,
        % child=2, immigrated=3)
        paramType

        maxValue % Larges possible bit sequence in decimal form
        type     % In what form the bit sequences are stored eg uint8
    end

    methods
        function obj = Generation(func, funcParamBits, mode)
            obj.c_func          = func;
            obj.c_funcParamBits = funcParamBits;
            obj.c_mode          = mode;

            % Find the smallest int that can hold the parameters
            if (funcParamBits <= 8)
                obj.type = @uint8;
            elseif (funcParamBits <= 16)
                obj.type = @uint16;
            elseif (funcParamBits <= 32)
                obj.type = @uint32;
            else
                obj.type = @uint64;
            end
            % Higherst posible paramet value, (in decimal reprenestation)
            obj.maxValue = 2^funcParamBits-1;
        end

        function migrateInto(obj, elements)
            newElements =   obj.randomParam(elements)'; % Make # new elements
            obj.inject(newElements, 3);                 % Add them to the generation
        end
        function mutatePopulation(obj, probability, safe, weightFunc)
            obj.sortParam();
            % How many are mutated
            mutations = round((length(obj.params)-safe)*probability*(rand+0.5));
            if mutations > 0
                % Indexes of the mutated elements
                idxPercent = weightFunc(rand(mutations,1));
                [~,idx] = sort(idxPercent);
                idx = idx + safe;
                obj.mutate(idx); % Change the parameter valie
                obj.paramVals(idx) = obj.evaluate(obj.params(idx)); % update the functional value
            end
        end
        function sortParam(obj)
            % Sort the parameters based on if we maximize or minimize
            [obj.paramVals, idx] = sort(obj.paramVals, obj.c_mode);
            obj.params =           obj.params(idx);
            obj.paramType =        obj.paramType(idx);
        end
        function values = getAsMatrix(obj)
            % Return the results as a 3 column matrix
            values = [obj.params', obj.paramVals', obj.paramType'];
        end
        function inject(obj, values, type)
            %  Add new (given) elements to the generation
            if (numel(values) > 0) % only add them if there is stuff to add
                if (nargin == 2)
                    type = 0;
                end
                obj.params =    [obj.params, values];       % Add them to the list
                obj.paramVals = [obj.paramVals, obj.evaluate(values)]; % Update function value
                obj.paramType = [obj.paramType, type*ones(size(values))]; % State there origin
            end
        end
        function gen = propagate(obj, weightFunc, combineFunc, nrChildren)
            % unweighted distribution of idexes (0 to 1)
            idxPercentNoWeight = (1:length(obj.params))/length(obj.params);
            % weighted distribution of indexes (0 to 1)
            idxPercent = weightFunc(idxPercentNoWeight);
            % convert (0 to 1) into actual indexes
            [~,idx] = sort(idxPercent);
            % matrix with [1  1  2  2  3  3 ...]
            pairs = ceil((1:length(obj.params))/2);
            % Total number of pairs
            lastPair = max(pairs);
            % Preallocate matrix for the resulting children
            children = zeros(lastPair*nrChildren,1, class(obj.params));
            for pair=1:lastPair
                % Get the indexes of the current pairs
                currentPairIdx = idx(pairs == pair);
                % Retreve the actual parameter values
                parents = obj.params(currentPairIdx);
                % Combine them to a child (with a passed function)
                i = (pair-1)*nrChildren+1; % start index in the children list
                children(i:i+nrChildren-1) = ...
                            combineFunc(parents, nrChildren);
            end
            % Make a new generation
            gen = Generation(obj.c_func, obj.c_funcParamBits, obj.c_mode);
            % Pass the children to the new generation
            gen.inject(children', 2);
        end
```

```
        function limitPop(obj, maxPopulation)
            % Remove last elements untim with maxPopulation
            obj.params = obj.params(1:maxPopulation);
            obj.paramVals = obj.paramVals(1:maxPopulation);
            obj.paramType = obj.paramType(1:maxPopulation);
        end
    end
    methods (Access=private)
        function param = randomParam(obj, count)
            if nargin==1
                count = 1; % default value for count
            end
            % Make $count random numbers between 0 and maxValue
            % In the proper uint format (func2str(obj.type) is like 'uint8')
            param = randi([0, obj.maxValue], [count,1], func2str(obj.type));
        end
        function values = evaluate(obj, values)
            values = obj.c_func(values);
        end
        function mutate(obj, idx)
            % Do a random bitflip in a parameter
            % Select what bit to flip
            bit =   randi([1, obj.c_funcParamBits], [1,length(idx)]);
            % Get the param bit sequence
            value = obj.params(idx);
            % Change the bit in the param
            obj.params(idx) = bitset(value, bit, ~bitget(value, bit));
            obj.paramVals(idx) = obj.evaluate(obj.params(idx));
        end
    end
end
```