## Environmental Selection

```
function [Population,FrontNo,CrowdDis] = EnvironmentalSelection(Population,N)
% The environmental selection of NSGA-II

    %% Non-dominated sorting
    [FrontNo,MaxFNo] = NDSort(Population.objs,Population.cons,N);
    Next = FrontNo < MaxFNo;

    %% Calculate the crowding distance of each solution
    CrowdDis = CrowdingDistance(Population.objs,FrontNo);

    %% Select the solutions in the last front based on their crowding distances
    Last     = find(FrontNo==MaxFNo);
    [~,Rank] = sort(CrowdDis(Last),'descend');
    Next(Last(Rank(1:N-sum(Next)))) = true;

    %% Population for next generation
    Population = Population(Next);
    FrontNo    = FrontNo(Next);
    CrowdDis   = CrowdDis(Next);
end
```

## polyMutateCore

```
function result = polyMutateCore(genome, lb, ub, eta)
% Ported from the polynomial mutation in Pymoo, circa early 2022

% This function is written by Ian Meyer Kropp

    delta1 = (genome - lb) / (ub - lb);    % Should be between 0 and 1
    delta2 = (ub - genome) / (ub - lb);    % Should be between 0 and 1

    exp = (eta + 1) ^ -1;

    ran = rand(size(genome));
    deltaq = zeros(size(genome));

    leftMask = ran < 0.5;
    rightMask = ran >= 0.5;

    xy = 1 - delta1;
    val = 2.0 * ran + (1.0 - 2.0 * ran) .* (xy .^ (eta + 1.0));
    d = (val .^ exp) - 1.0;
    deltaq(leftMask) = d(leftMask);

    xy = 1.0 - delta2;
    val = 2.0 * (1.0 - ran) + 2.0 * (ran - 0.5) .* (xy .^ (eta + 1.0));
    d = 1.0 - (val .^ exp);
    deltaq(rightMask) = d(rightMask);

    muted_genome = genome + deltaq .* (ub - lb);

    muted_genome = min(max(muted_genome,lb),ub);

    result = muted_genome;
end
```

## sbx

```
function Offspring = sbx(Parent, lb, ub, Parameter)
% Migrated from the PlatEMO OperatorGA module for convenience

% This function is written by Ian Meyer Kropp

    [proC,disC] = deal(Parameter{:});

    Parent1 = Parent(1:floor(end/2),:);
    Parent2 = Parent(floor(end/2)+1:floor(end/2)*2,:);
    [N,D]   = size(Parent1);

    beta = zeros(N,D);
    mu   = rand(N,D);
    beta(mu<=0.5) = (2*mu(mu<=0.5)).^(1/(disC+1));
    beta(mu>0.5)  = (2-2*mu(mu>0.5)).^(-1/(disC+1));
    beta = beta.*(-1).^randi([0,1],N,D);
    beta(rand(N,D)<0.5) = 1;
    beta(repmat(rand(N,1)>proC,1,D)) = 1;
    Offspring = [(Parent1+Parent2)/2+beta.*(Parent1-Parent2)/2
                 (Parent1+Parent2)/2-beta.*(Parent1-Parent2)/2];

    Lower = repmat(lb,2*N,1);
    Upper = repmat(ub,2*N,1);

    % Put everything back in bounds
    Offspring = min(max(Offspring,Lower),Upper);
end
```

## sm2target

```
function newPop = sm2target(Pop, lb, ub, newSparsities)

% This function is written by Ian Meyer Kropp

    % Sparse Mutate to a target
    if numel(Pop) == 0
        newPop = Pop;
        return;
    end

    [~,D] = size(Pop);

    sparsities = sum(Pop == 0, 2) / D;

    mutateMask = newSparsities ~= sparsities;

    indv2mut = find(mutateMask);

    % determine the non-zero increase/decrease for each indiv
    nz2add = round(D * (sparsities(mutateMask) - newSparsities(mutateMask)));

    % find where the non-zeros are
    [zIndvs,   zGenes] = find(Pop(indv2mut,:) == 0);
    % find where the zeros are
    [nzIndvs, nzGenes] = find(Pop(indv2mut,:) ~= 0);

    newNzs = false(size(Pop));
    newZs = false(size(Pop));


    for i = 1:size(indv2mut,1)
        % gather relevant info
        indv_i = indv2mut(i);

        % Case where more non-zeros are needed
        if nz2add(i) > 0
            % find where there are zeros to flip
            zeroLocs = zGenes(zIndvs == i);

            % determine how many of them to flip
            numToFlip = nz2add(i);

            % Determine which of these posible zero positions to flip
            toFlip = zeroLocs(randperm(length(zeroLocs), numToFlip));

            % Record these positions
            newNzs(indv_i, toFlip) = true;

        % Case where more zeros are needed
        else

            % find where there are non-zeros to flip
            nZeroLocs = nzGenes(nzIndvs == i);

            % determine how many of them to flip
            numToFlip = -nz2add(i);

            % Determine which of these posible non-zero positions to flip
            toFlip = nZeroLocs(randperm(length(nZeroLocs), numToFlip));

            % Record these positions
            newZs(indv_i, toFlip) = true;
        end
    end

    %% Make the mutations
    % Find the min/max of the genome positions to mutate
    [~, newNzsCols] = find(newNzs);

    newNzsLb = lb(newNzsCols);
    newNzsUb = ub(newNzsCols);

    Pop(newNzs) = newNzsLb + rand(1,sum(newNzs, 'all')) .* (newNzsUb - newNzsLb);
    Pop(newZs) = zeros(sum(newZs, 'all'), 1);

    newPop = Pop;
end
```

## SNSGAII

```
classdef SNSGAII < ALGORITHM
% <multi> <real> <large/none> <constrained/none> <sparse>
% Sparse nondominated sorting genetic algorithm II

% This function is written by Ian Meyer Kropp

    methods
        function main(Algorithm, Problem)

            [ sampling_method, mutation_method, crossover_method ] = ...
               Algorithm.ParameterSet( ...
                 {@vssps, 0.75, 1}, ...
                 @spm, ...
                 @ssbx ...
               );

            %% Generate random population

            sampler = sampling_method{1};
            lowerBound = sampling_method{2};
            upperBound = sampling_method{3};
            Population = sampler(Problem, lowerBound, upperBound);

            [~,FrontNo,CrowdDis] = EnvironmentalSelection(Population,Problem.N);

            %% Optimization
            while Algorithm.NotTerminated(Population)
                MatingPool = TournamentSelection(2,Problem.N,FrontNo,-CrowdDis);

                Offspring  = sparseOperatorGA(Problem, Population(MatingPool), ...
{1,20,1,20,1,20,mutation_method,crossover_method});
```

```
                [Population,FrontNo,CrowdDis] = 
EnvironmentalSelection([Population,Offspring],Problem.N);
            end
        end
    end
end
```

## sparseOperatorGA

```
function Offspring = sparseOperatorGA(Problem, Parent, Parameter)
% Adapted from OperatorGA in PlatEMO by Ian Meyer Kropp

% This function is written by Ian Meyer Kropp

    %% Parameter setting
    if nargin > 1
        [proC,disC,proM,disM,proSM,disSM,mutation_method,crossover_method] = 
deal(Parameter{:});
    else
        [proC,disC,proM,disM,proSM,disSM,mutation_method,crossover_method] = 
deal(1,20,1,20,true,true);
    end

    calObj = false;

    if isa(Parent(1),'SOLUTION')
        calObj = true;
        Parent = Parent.decs;
    end

    % Check if any of the decision variables are non-real values
    if any(ones(size(Problem.encoding)) ~= Problem.encoding)
        error('Only real encoding supported.');
    end

    Offspring = crossover_method(Parent, Problem.lower, Problem.upper, 
{proC,disC});

    mutation_params = {proM,disM, proSM,disSM};

    Offspring = mutation_method(Offspring, Problem.lower, Problem.upper, 
mutation_params);

    if calObj
        Offspring = Problem.Evaluation(Offspring);
    end
end
```

## spm

```
function newPop = spm(Pop, lb, ub, Parameter)

% This function is written by Ian Meyer Kropp

    % Each row is a different population member
    % Each column is a different genome

    if nargin > 3
        [probMut,distrMut, probSMut, distrSMut] = deal(Parameter{:});
    else
        [probMut,distrMut, probSMut, distrSMut] = deal(1,20,1,20);
    end

    [N,D] = size(Pop);

    % Determine where the zeros are
    nonZeroMask = Pop ~= 0;

    %% Value mutations
    ran = rand(size(Pop(nonZeroMask)));

    toMutateNZ = ran < (probMut/D);

    toMutate = false(size(Pop));

    toMutate(nonZeroMask) = toMutateNZ;

    [~, genomesToMutate] = find(toMutate);

    lb_pm = lb(genomesToMutate)';
    ub_pm = ub(genomesToMutate)';

    % mutate values
    Pop(toMutate) = polyMutateCore(  Pop(toMutate), ...
                                    lb_pm, ub_pm, distrMut);

    %% Sparsity mutations

    % Determine which population members to mutate sparsity
    ran = rand(N, 1);

    mutateMask = ran < probSMut/D;

    % Figure out the individual sparsities of each individual
    sparsities = sum(Pop == 0, 2) / D;

    lb_sp = zeros(sum(mutateMask), 1);
    ub_sp = ones(sum(mutateMask), 1);

    newSparsities = sparsities;

    newSparsities(mutateMask) = polyMutateCore(sparsities(mutateMask), lb_sp,
ub_sp, distrSMut);

    newSparsities = min(max(newSparsities,0),1);

    % check if there's anything to do
    if newSparsities == sparsities
        newPop = Pop;
    else
        newPop = sm2target(Pop, lb, ub, newSparsities);
    end
end
```

## sssbx

```
function newPop = ssbx(Parent, lb, ub, Parameter)

% This function is written by Ian Meyer Kropp

    %% Fetch paramters/setup
    if nargin > 3
        [proC,disC] = deal(Parameter{:});
    else
        [proC,disC] = deal(1,20);
    end

    Parent1 = Parent(1:floor(end/2),:);
    Parent2 = Parent(floor(end/2)+1:floor(end/2)*2,:);

    % figure out where are zeros/non-zeros
    zMaskP1 = Parent1 == 0;
    zMaskP2 = Parent2 == 0;

    nzMaskP1 = ~zMaskP1;
    nzMaskP2 = ~zMaskP2;

    % figure out which positions are both zero or both non-zero
    matching = (zMaskP1 & zMaskP2) | (nzMaskP1 & nzMaskP2);
    not_matching = ~matching;

    % empty template for results
    Offspring1 = ones(size(Parent1))*-99;
    Offspring2 = ones(size(Parent2))*-99;

    %% Step 1: crossover on positions are both non-zero or both zero

    [~,genes2sbx] = find(matching);

    sbx_results = sbx([Parent1(matching)';Parent2(matching)'], lb(genes2sbx),
ub(genes2sbx), {proC, disC});

    Offspring1(matching) = sbx_results(1,:)';
    Offspring2(matching) = sbx_results(2,:)';

    %% Step 2: swap values that are mismatches between zero and non-zero

    % empty mask of which positions to swap
    swap_mask = ones(sum(not_matching, 'all'), 1);

    % generate random number to determine how many zeros/non-zeros will go
    % to each child
    z2nzRatio = unifrnd(0,1);

    swap_mask = sm2target(swap_mask', 0, 1, z2nzRatio);

    swap_mask = swap_mask == 1;

    % swap
    not_matching_p1 = Parent1(not_matching);
    not_matching_p2 = Parent2(not_matching);

    not_matching_p1_temp = not_matching_p1;
    not_matching_p1(swap_mask) = not_matching_p2(swap_mask);
    not_matching_p2(swap_mask) = not_matching_p1_temp(swap_mask);

    Offspring1(not_matching) = not_matching_p1;
    Offspring2(not_matching) = not_matching_p2;

    % return result
    newPop = [Offspring1; Offspring2];
end
```

## vssps

```
function Population = vssps(prob, sLower, sUpper)
% Randomly generate an initial population

% This function is written by Ian Meyer Kropp

    %  Nomenclature example
    %  N = 8
    %  D = 14
    %
    %          Cycle length of 14
    %          |
    %  |-------|-----------------|
    %  1 1 1 1                        -
    %          1 1 1 1                |-- One full cycle
    %                  1 1 1          |
    %                        1 1 1    -
    %     |-------|------|-----|---------------- Cycle count of 4
    %   |---|--|-|----------------------Cycle count of 4
    %  1 1
    %      1 1
    %         1
    %           1
    %  |----|----|
    %       |
    %      Cyle length of 6

    %% Result set up
    pop = prob.Initialization();
    varCount = size(prob.lower,2);
    mask = false(prob.N, varCount);

    %% Determine the positioning of each stripe per individual
    densityVector = 1 - linspace(sLower, sUpper, prob.N);

    widthVector = round(densityVector.*prob.D);

    % Put widths back into bound if rounding error occurred
    lb = floor((1- sLower)*prob.D);
    widthVector(widthVector > lb) = lb;

    cumulativeWidths = cumsum(widthVector);

    % if all sparsities are 100%, then skip processing, since everything
    % will be zeros
    if sum(widthVector == 0) == prob.N
        processedIndvs = prob.N;
    else
        processedIndvs = 0;
    end

    cycle_count = 0;
    cycles = zeros(prob.N, prob.D);
    while processedIndvs < prob.N
        % Figure out how many stripes will fit in this cycle
        cycle_count = cycle_count + 1;
        spotsThatFitMask = cumulativeWidths <= prob.D & cumulativeWidths ~= 0;
        numThatFit = sum(spotsThatFitMask);
        largestFit = max(cumulativeWidths(spotsThatFitMask));

        cumulativeWidths = cumulativeWidths - largestFit;
        cumulativeWidths(cumulativeWidths<0) = 0;
        processedIndvs = processedIndvs + numThatFit;
        spotsThatFit = find(spotsThatFitMask);
        cycles(cycle_count,1:numThatFit) = spotsThatFit;

    end

    %% Create density mask

    % Mask out non-zero values cycle-by-cycle
    currentIndv = 1;
    for c = 1:cycle_count
        cycle = cycles(c, cycles(c, :) ~= 0);
```

```
            widths = widthVector(cycle);

            gapToFill = prob.D - sum(widths);
            gapSize = ceil((prob.D - sum(widths))/numel(widths));
            % For each individual in the cycle
            position = 1;
            for i = 1:numel(widths)

                width = widths(i);

                % Determine if a gap is needed
                gapWidth = 0;
                if gapToFill > 0
                    gapWidth = gapSize;
                    gapToFill = gapToFill - gapWidth;
                end

                % Determine the position of the stripe
                startPoint = position;

                if c == cycle_count
                    endPoint = position+width-1;
                else
                    endPoint = position+width-1+gapWidth;
                end

                % Prevent overflow from a gap calculation
                if endPoint > prob.D
                    endPoint = prob.D;
                end

                % Mask out stripe
                mask(currentIndv, startPoint:endPoint) = true;

                % Go to the next individual

                position = position+width+gapWidth;
                %position = position+width;

                currentIndv = currentIndv + 1;

            end

        end

        %% Mask off population according to stripe position
        sparse_pop = pop.decs;
        sparse_pop(~mask) = 0;

        % Recalculate objective and constraints
        popDec = prob.CalDec(sparse_pop);
        popObj = prob.CalObj(sparse_pop);
        popCon = prob.CalCon(sparse_pop);
```

```
    Population = SOLUTION(popDec, popObj, popCon);
end
```