Research Note
RN/13/09

Efficiently Vectorized Code for Population Based Optimization Algorithms

2013-03-28

Oliver Rice

Rickard Nyman

Abstract

This article outlines efficient vector code commonly required for population based optimization methods. Specifically, techniques for population generation, probabilistic selection, recombination & mutation are introduced. These coding ’best practices’ emphasize execution speed and concision over readability. As such, each snippet is initially coded in verbose, readable form, and subsequently condensed to maximize efficiency gains. Examples are provided in MATLAB code, though many port directly to other vector/matrix based languages such as R and Octave with minor syntactic adjustments.

1. Introduction

Population based optimization methods are most often associated with discrete optimization problems too large or complex to be solved deterministically. We focus primarily on the model of genetic algorithms though much of the proposed code is directly transferable to other algorithm candidates. These methods rely on generation of a randomly seeded population of solution candidates which are probabilistically selected for recombination and subsequently mutated. As this article is intended for practitioners with a general understanding of genetic algorithm structure, detailed theoretical explanations have been omitted.

This document outlines the basic components of genetic algorithms with MATLAB code samples. The code is initially presented using the typical C style approach within MATLAB, and then be condensed to efficient MATLAB code. Explanations are provided to detail sources of efficiency gains when possible.

General rules of thumb when writing in vector or matrix based programming languages are to avoid loops, leverage vector overloaded functions and utilize indexing to the maximal extent possible. While avoiding loops has become somewhat less crucial in recent years due to implementation of MATLAB's Just-In-Time (JIT) compiler replacing the previous fully interpreted architecture, other vector languages such as R and Octave remain fully constrained.

There are 5 sections in the most basic genetic algorithms. These sections are:

1. Initial Population
2. Fitness
3. Selection
4. Recombination
5. Mutation

With the exception of fitness, which is domain specific, each of these sections is presented with corresponding options for common population types.

2. Initial Population

Initial populations are generally seeded randomly. There are several common population types which are reviewed. Namely, random boolean, skewed boolean, random integer, user-defined integer distribution, and random permutations. A constant notation to be used throughout the document states that `N` describes the number of individuals in the population and `G`, the genome length. The goal in the population construction phase is to return a population (`Pop`) such that each row contains a genome. This implies `Pop` is an `NxG` matrix.

```
% Parameters
N = 100;                                    % Individuals in Population
G = 30;                                     % Genome Length
```

2.1. Random Boolean

Random boolean or logical populations are one of the most common and straight forward.

Example $\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}$ with `N = 3` and `G = 4`.

```
1  % C Code Equivalent: Random Boolean
2  for i = 1:N
3      for j = 1:G
4          if rand(1) ≥ 0.5                 % Split at midpoint
5              Pop(i,j) = 1;
6          else
7              Pop(i,j) = 0;
8          end
9      end
10 end
```

Lines 4-8 can be replace through employment the `round()` command which rounds inputs the the nearest integer.

```
1  % Step 1: Random Boolean
2  for i = 1:N
3      for j = 1:G
4          Pop(i,j) = round(rand(1));
5      end
6  end
```

Since both `round()` and `rand()` are overloaded such that `rand([N,G])` produces an `N` by `G` matrix with pseudo-random doubles between 0 and 1(inclusive) and `round(rand([N,G]))` performs element wise operations on either vectors or matrices, both `for` loops can be removed.

```
1  % Efficient: Random Boolean
2  Pop = round(rand([N,G]));
```

2.2. Skewed Boolean

This population is identical to random boolean population except that the distribution of genes is forced to a predefined percentage of 1's and 0's. Skewing the distribution within a population can be useful for accelerating convergence when the user has knowledge of the search space.

```
1  % C Code Equivalent: Skewed Boolean
2  PerOnes = 0.2                            % Percentage of Ones
3  for i = 1:N
4      for j = 1:G
5          if rand(1) ≥ (1−PerOnes)
```

```
            Pop(i,j) = 1;
        else
            Pop(i,j) = 0;
        end
    end
end
```

As we know `PerOnes` is bound between 0 and 1 a simple transform can be applied allowing the `round()` function to remain effective.

```
% Efficient: Random Boolean
PerOnes = 0.2
Pop = round(rand(N,G)+(PerOnes-.5));
```

This operation adds the deviation from the default rounding limit of 0.5 to the random matrix prior to rounding. Since `PerOnes` is always between 0 and 1 (exclusive), the maximum of each element's range in `(rand(N,G)+(PerOnes-.5))` is between -0.5 and 1.5 (exclusive) which remains within the desired boolean range when rounded.

2.3. Random Integers

At this point the combinatorial discrete boolean case is extended to positive integers. In this purely random example, each integer receives a percentage allocation equal to `1/MaxVal`.

```
% C Code Equivalent: Random Integers
MaxVal = 15;                            % Maximum Gene Value
for i = 1:N
    for j = 1:G
        randval = rand(1)*(MaxVal-1)+1;
        Pop(i,j) = round(randval);
    end
end
```

Array indexing in MATLAB begins at 1 instead of the more usual zero. For efficient use later on it is often beneficial to have genome integers match a separate array used in genome fitness evaluation. By decrementing MaxVal by 1 and adding 1 to the entire random matrix we enforce a minimum and maximum genome value between 1 and MaxVal (inclusive). This is preferable as it avoids repetitive index adjustment in coming steps.

```
% Efficient: Random Integers
MaxVal = 15;                            % Maximum Gene Value
Pop = round(rand(N,G)*(MaxVal-1)+1);
```

As before, each loop can be compounded by applying overloaded functions.

2.4. Random Integers with Predefined Distribution

Welcome to the first non-trivial problem. In this example a population consists of indexable (1 to `MaxVal`) integers with a user-defined distribution. The user-defined distribution `D` is entered as a percentage of the population allocated to each integer, in order. To function correctly `D` must sum to 1.

```
1  % C Code Equivalent: Random Integers - User Defined Distribution
2
3  D  = [.1, .5, .08, .2, .12];       % User-Defined Distribution
4
5  for i = 1:length(D)                % Begin: Compute Cumulative Distribution
6      if i ==1                       %.
7          CD(i) = D(i);              %...
8      else                           %.....
9          CD(i) = CD(i-1) + D(i);    %...
10     end                            %.
11 end                                % End: Compute Cumulative Distribution
12
13 MaxVal= length(CD);                % Maximum Integer Value
14 for i = 1:N
15     for j = 1:G
16         Pop(i,j) = rand(1);
17         for k = 1:MaxVal
18             if k == 1
19                 if Pop(i,j) ≥0 & Pop(i,j)≤CD(k) % Bin rand vals to integers
20                     Pop(i,j) = k;
21                 end
22             else
23                 if Pop(i,j)≥CD(k-1) & Pop(i,j)≤CD(k) % Catch  remaining
24                     Pop(i,j) = k;
25                 end
26             end
27         end
28     end
29 end
```

Lines 5 to 11 can be swapped for built in function `cumsum(D)` which computes the row-wise cumulative sum of input vectors and matrices.

```
% Cummulative Sum
CD  = horzcat(0,cumsum([.1, .5, .08, .2, .12]));
```

The next loops in Lines 14 to 28 'bin' the data between the values listed within the cumulative sum variable `CD`. This procedure can not be fully vectorized without creating as many copies of the population matrix as there are bins. When population size, genome length, and/or number of bins are numerous, creating multiple copies of the population can exhaust RAM resources and cause writing to the hard drive. If this occurs, the process is likely to stop responding. That said, it is possible to eliminate 2 of the 3 `for` loops without limiting hardware compatibility. Given the overhead associated with large loops we prefer to leave the third, or comparison loop in code while vectorizing the other two.

```
% Step 1: Random Integers - User Defined Distribution
Pop = rand(N,G);                     % Random values
D  = [.1, .5, .08, .2, .12];         % User Distribution
CD = cumsum(D);                      % Cumulative Sum of Distribution
MaxVal = size(D,2);                  % Max Value of Integers
```

```
 6  Cat = ones(size(Pop));                  % Categorization Variable
 7  for i = 1:MaxVal−1
 8      Cat = Cat + double(Pop>CD(i));      % Bin to Categories
 9  end
10  Pop = Cat;                              % Set Pop equal to Categories
```

A new variable `Cat` the same size as `Pop` is created. At initialization, Line 6, all values are set to 1. In other words, the default assumption states that all randomly generated values fall within the first and second values in the cumulative sum. The remaining `for` loop tests each element in the population matrix against the relevant position in the cumulative distribution. By testing `Pop>CD(i)` at each stage of the loop and incrementing values when true, the gene values increase within `Pop` until they match the correct location within `CD`. Note that the comparison of `Pop` and `CD` in line 8 results with a logical, or index matrix. To convert the logicals to a numeric matrix for use in addition to `Cat`, it is converted using `double()`.

Once the structure is apparent, a condensed version can be found below.

```
1  % Efficient Looped: Random Integers − User Defined Distribution
2  Pop = rand(N,G);                        % Random values
3  CD  = horzcat(0,cumsum([.1, .5, .08, .2, .12])); % CumSum of Distribution
4  MaxVal = size(CD,2)−1;                  % Max Value of Integers
5  for i = MaxVal:−1:1
6      Pop(Pop≥CD(i) & Pop≤1) = i;         % Bin to Population
7  end
```

There are three main differences with this approach. First is the cumulative distribution. The distribution is computed directly from inputs rather than using a second variable. For ease of reference a 0 is horizontally concatenated (`horzcat()`) to the cumulative variable. Second, in order to avoid creating a new `Cat` variable the loop runs in descending order. This is necessary as the second comparison on Line 6 (`Pop≤1`) is a requisite to preventing re-categorization of random values which have already been binned, based on their bin index. This condition could be omitted, thus allowing for the loop to run in ascending order but it would not be robust to the possibility of the pseudo-random number generator yielding a value of 1 anywhere in the population.

For users with sufficient RAM interested in performance gains, the entire process can be vectorized as below.

```
% Step 1 Vectorized: Random Integers − User Defined Distribution
Pop    = rand(N,G);                      % Random values
CD     = cumsum([.1, .5, .08, .2, .12]); % CumSum of Distribution
MaxVal = size(CD,2);                     % Maximum Integer Value
CDa    = reshape(CD,[1,1,MaxVal]);       % Pre−process distribution
CDR    = repmat(CDa,[N,G,1]);            % Replicate distribution
PopR = repmat(Pop,[1,1,MaxVal]);         % Replicate Population
idx    = PopR>CDR;                       % Compare 3−D matricies
[¬,Pop]= min(idx,[],3);                  % Location of 1st 0 is Correct Bin
```

The first difference encountered is in Line 5. By reshaping the vector it is projected into 3-dimensional space. By then applying the function `repmat()` with inputs `[N,G,1]`, the cumulative distribution is copied N times in the first dimension and G times in the second dimension. The resulting matrix is of size `NxGxlength(CD)`. In order to apply native indexing capabilities, `Pop` must then be replicated `length(CD)` times to be equal in size to the cumulative distribution object. This activity is performed in Line 7. A direct $>$ comparison is then made in Line 8 which compares each random value in the population with each value in the cumulative distribution. We next locate the index of the first occurrence when the population value is not greater than the cumulative distribution. This index corresponds to the correct 'bin' for the given random number. The 3-D indexing approach can also be represented in a more concise form.

```
% Efficient Vectorized: Random Integers — User Defined Distribution
CD    = cumsum([.1, .5, .08, .2, .12]);          % CumSum
MaxVal = size(CD,2);                             % Max Integer Value
[¬,Pop] = min(repmat(rand(N,G),[1,1,MaxVal])> ...
    repmat(reshape(CD,[1,1,MaxVal]),[N,G,1]),[],3); % Binning
```

2.5. Random Permutations

Permutations are another example of a common genome type in population based optimization algorithms. In this section a permutation set begins with 1 and ends with `MaxVal`. Since genome length `G` is fixed, `MaxVal = G`.

Needless to say, there are many ways of producing permutations in C code. Efficiency is largely dependent on size of `G` and none of the most efficient means are easily readable. Instead, we simply initialize each genome as an ordered list 1 to G. Once initialized, random locations on the genome are swapped until the output is a randomized permutation.

```
% Pseudo — C Code Equivalent: Random Permutations

for i = 1:N
    for j = 1:G
        Pop(i,j) = j;                   % Random value
    end
    for j = 1:4*G                       % Arbitrarily large loop maximum
        Loc1 = round(rand(1)*(G—1)+1);  % Location of First Swap Point
        Loc2 = round(rand(1)*(G—1)+1);  % Location of Second Swap Point
        Holder = Pop(i,Loc1);           % Hold Value to be Replaced
        Pop(i,Loc1) = Pop(i,Loc2);      % First Half of Swap
        Pop(i,Loc2) = Holder;           % Second Half of Swap
    end
end
```

Based on past examples, the obvious choice is to search for a built in function overloaded to produce `N` random samples of permutations with length `G`. The closest match is `randperm()`. While `randperm()` does accept permutation length `G`, it does not allow for multiple samples from a single function call. As such, the function must be called `N` times to produce an entire population.

```
1  % Loop: Random Permutations with Loops
2  for i = 1:N
3      Pop(i,:) = randperm(G);
4  end
```

Depending on system architecture & software version it may be faster to issue the command in parallel using a `parfor` loop or `arrayfun()` with a function handle.

```
1  % Option 2,: Random Permutations with Function Handle
2  matlabpool                        % Creates MATLAB 'workers': 1 per CPU
3  
4  parfor i = 1:N
5      Pop(i,:) = randperm(G);
6  end
```

The `parfor` or 'parallel for loop' first creates a MATLAB worker process for each available CPU. Once the local 'cluster' has been launched, the `parfor` tag dictates that each iteration of the for loop is strictly independent. This allows each CPU in the cluster to dynamically process iterations in parallel and theoretically improve execution speed. Another parallel approach employs `arrayfun()`.

```
1  % Option 3, Step 1: Random Permutations with Function Handle
2  Pop = arrayfun(@(x) randperm(G), 1:N, 'UniformOutput', false);
```

The `arrayfun()` function eliminates the need to launch a cluster of worker processes to execute code. Instead, this method launches each iteration of the function as a separate event. One disadvantage to this approach is the 'cell array' return type though this can easily be converted to the previous matrix of doubles using `cell2mat()`.

```
1  % Option 3: Random Permutations with Function Handle
2  Pop = cell2mat(arrayfun(@(x) randperm(G), 1:N, 'UniformOutput', false)');
```

Notice the `'` signifying a transpose as the third to last character in Line 2. This addition coerces the cell array to convert to a matrix as opposed to an appended vector. It is worth reiterating that the speed of each methodology is highly dependent on the target system architecture. Variations in execution time are likely to be orders of magnitude for each approach on any given system architecture.

This tangent into parallelization may be useful when generating custom genome representations but none of these are most efficient for producing multiple samples of random permutations. While the `randperm()` function is not capable of producing more than 1 permutation at a time, we can apply a work around using the `sort()` function. The intended use of `sort()` returns a sorted listed of input elements. However, it also produces a second output containing the inputs' original order index. When the input matrix is randomly generated, this second output is equivalent to a random permutation.

```
1  % Efficient: Random Permutations
```

```
[¬,Pop] = sort(rand(N,G),2);
```

3. Selection

While efficiently producing an initial random population is a best practice, the code is run only once per execution of the algorithm. In contrast the fitness, selection, recombination, and mutation procedures are executed once with every generation. Efficiently coding these stages will have a much more significant impact on reducing total execution time than the initial population.

By the selection stage the user is expected to have solved for a row vector of fitness values `F` of size `Nx1`. The three types of selection to be demonstrated are Roulette Wheel, Tournament, and 50% Truncation. Each function produces a `2*Nx1` output returning the indices of each 'winning' genome in row vector `W`.

3.1. Roulette Wheel

Roulette wheel selection draws each solution for recombination from the entire population. The probability of each solution i being selected is defined as:

$$SelectionProbability = \frac{Fitness(i)}{Sum(Fitness(:))}$$

In MATLAB equivalent C code:

```
% C Code Equivalent: Roulette Wheel Selection
for i = 1:N                             % Begin: Compute Cumulative Distribution
    if i ==1                            %.
        FCD(i) = F(i);                  %...
    else                                %.....
        FCD(i) = FCD(i-1) + F(i);%...
    end                                 %.
end                                     % End: Compute Cumulative Distribution

for i = 1:N
    FCD(i) = FCD(i)/FCD(end);           %Normalize Cumulative Distribution
end

for i = 1:2*N
    randval = rand(1);
    for k = 1:N
        if k == 1
            if randval ≥0 && randval≤FCD(k)
                W(i) = k;
            end
        else
            if randval≥FCD(k-1) && randval≤FCD(k)
                W(i) = k;
            end
        end
    end
end
```

Once again, the Lines 4 to 10 can be replaced with the `cumsum()` function.

```
1  %% Step 1: Roulette Wheel
2  FCD = cumsum(F);              % Fitness Cumulative Distribution
3  FCD   = FCD/FCD(end);         % P(Selection) by Row
4  Fa   = FCD*ones(1,2*N);       % Replicating P(Selection) for Comparison)
5  R    = rand(1,2*N);           % Random Roulette Spins 0<X<1
6  Ra    = ones(N,1)*R;          % Replicate Random Values
7  idx = Ra>Fa;                  % Logical Index of Value
8  [¬,W] = min(idx,[],1);        % First 0 Index is the Winner
```

Since there is no guarantee that the sum of all fitness values is 1, the cumulative distribution is divided through by its maximum element in Line 3. To enable simultaneous comparison, the cumulative distribution is replicated `2*N` times through multiplying by an appropriately sized matrix of ones. Line 5 'spins the roulette wheel' `2*N` times. The result is then replicated across `N` rows. Given `Ra` and `Fa` are then equal in size, they can be directly compared. The first instance of value 0 returns the index of the winning individual in the population. In abridged form:

```
%% Efficient: Roulette Wheel
[¬,W] = min(ones(N,1)*(rand(1,2*N))>((cumsum(F)*ones(1,2*N)/sum(F))),[],1);
```

3.2. Tournament

Tournament selection is a widely popular methodology which probabilistically selects genomes for recombination based on fitness. A key attribute of this type is that it maintains genome diversity more robustly than competitors when a small percentage of the population is significantly more fit than average. The algorithm randomly selects genomes to 'compete' from the population in 'tournaments' of user selected size size `S`. The genome yielding highest fitness in of each tournament is selected for recombination. Some variants also select the second best genome in each tournament for recombination to halve the number of necessary cycles. The demonstrated approach provides a single winner per tournament.

```
%% C Code Equivalent: Tournament Selection
S = 6;                              % Tournament Size
for i = 1:2*N
    for j = 1:S
        T(i,j)  = round(rand(1)*(N−1)+1); % Add Genome to Tournament
    end
    Mx = 0;                         % Max Fitness In Tournament
    for j = 1:S
        if F(T(i,j))>Mx             % If Genome is Better than Current Best
            Mx    = F(T(i,j));      % Reset Current Best
            W(i)  = T(i,j);         % Update Winner
        end
    end
end
```

Lines 4 to 6 can be replaced with the previously described method for producing random integers to a maximum value. In this case the maximum value must be `N`.

```
%% Efficient: Tournament Selection
S  = 6;                                % Tournament Size
T  = round(rand(2*N,S)*(N-1)+1);       % Tournaments
[¬,idx] = max(F(T),[],2);              % Index to Determine Winners
W  = T(sub2ind(size(T),(1:2*N)',idx)); % Winners
```

We then use each tournament's genome indices to assemble the fitness of each tournament's participants. By taking a row wise `max()` of the assembly, the row index of each winner is given as secondary output. The function `sub2ind()` converts the row wise indexes of each tournament winner to an index which retrieves the population row index of each tournament's winning genome.

3.3. 50% Truncation

During 50% truncation selection the best 50% of genomes are selected for recombination and the lower half are eliminated from the pool of candidates. C code equivalent is omitted in this case due to the straightforward nature of the problem and the complexity of a sorting routine without use of functions.

```
1 %% Efficient: 50% Truncation
2 [¬,V] = sort(F);                            % Sort Fitness in Ascending Order
3 V = V(N/2+1:end);                           % Winner Pool
4 W = V(round(rand(2*N,1)*(N/2-1)+1))';% Winners
```

Line 2 sorts the fitness values `F` in ascending order and saves the original index output in variable `V`. Half of the population with lowest fitness are eliminated from the pool of possible winners in Line 3. From the remaining pool of possible winners `2*N` samples are chosen without bias.

4. Recombination

Recombination or crossover is domain specific by population type. That said, there are several common recombination techniques for the population types we have defined. Specifically, uniform, single point, and double point crossover for combinatorial problems, and single point preservation for permutation problems. The target output of this phase is a new population `Pop2` containing elements of the vector of winners `W`. It is taken that each unique adjacent pair of genome indices in `W` is are selected for recombination e.g. if `W = [11, 7, 13, 41, 2, 23]` the resulting pairs are `[11,7], [13,41]`, and `[2,23]`.

4.1. Uniform

This crossover type is primality intended for problems where each element in the genome is independent of the others. In other words, relative location of genes to each other the in genome has no baring on fitness. Uniform crossover creates an solution in the `t+1` generation by randomly selecting genes from each of the selection winners corresponding to the relevant position.

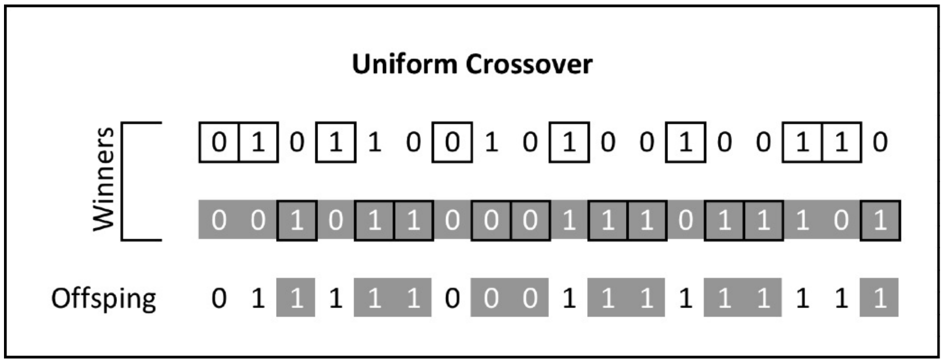

Figure 1: Uniform Crossover Example with Boolean Population

```
%% C Code Equivalent : Uniform Crossover
for i = 1:N
    for j = 1:G
        randval = round(rand(1));       % Parent 1 or Parent 2 (0 or 1)
        idx     = (i-1)*2+1+randval;    % Build index from W vector
        Pop2(i,j) = Pop(W(idx),j);      % Add Gene to Genome
    end
end
```

```
%% Efficient: Uniform Crossover
idx  = logical(round(rand(size(Pop)))); % Index of Genome from Winner 2
Pop2 = Pop(W(1:2:end),:);               % Set Pop2 = Pop Winners 1
P2A  = Pop(W(2:2:end),:);               % Assemble Pop2 Winners 2
Pop2(idx) = P2A(idx);                   % Combine Winners 1 and 2
```

First, an index is created with 50% 0s and 50% 1s in random order. Next `Pop2` or, the next population, is set equal to the genome values of all 'first' selection winners. These winners are defined by index `1:2:end` which equates to `[1, 3, 5, 7, ...]`. Next, a holder variable `P2A` is set equal to the second set of selection winners. Finally, the logical index is used to replace elements in `Pop2` with the equivalently located element of `P2A` in all locations where `idx == 1`.

4.2. One-Point

Crossing selection winners or 'parent' genomes at a single randomly selected location is less destructive than uniform crossover for problems where adjacency of genes is at least partially determinant of genome fitness.

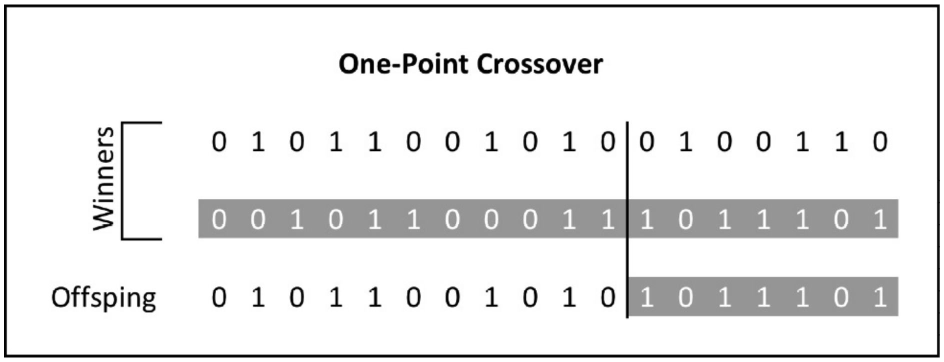

Figure 2: One-Point Crossover Example with Boolean Population

```
 1  %% C Code Equivalent : One-Point Crossover
 2  for i = 1:N
 3      CP = round(rand(1)*(N-1)+1);          % Genome Crossover Point
 4      for j = 1:G
 5          if j<CP
 6              idx = (i-1)*2+1;              % Build index from W vector
 7          else
 8              idx = (i-1)*2+2;              % Build index from W vector
 9          end
10          Pop2(i,j) = Pop(W(idx),j);        % Add Gene to Genome
11      end
12  end
```

The C equivalent code is relatively straight forward. In Line 3 a crossover point is selected. A conditional statement then pulls all genes with index values less than the crossover point from the relevant winner 1 and remaining genes from winner 2.

```
1  %% Efficient: One-Point Crossover
2  Pop2 = Pop(W(1:2:end),:);                  % Set Pop2 = Pop Winners 1
3  P2A  = Pop(W(2:2:end),:);                  % Assemble Pop2 Winners 2
4  Ref  = ones(N,1)*(1:G);                    % Reference Matrix
5  idx  = (round(rand(N,1)*(G-1)+1)*ones(1,G))>Ref; % Logical Index
6  Pop2(idx) = P2A(idx);                      % Recombine Winners
```

To efficiently vectorize the comparison each winner set is assembled into a population matrix. A reference matrix is then built to contain repeated rows of column indices. By comparing random integers between `1` and `G` to the column index matrix we receive a logical index which is split randomly in one location between series of 0s and 1s. This logical matrix can be used to index crossover in Line 6.

4.3. Two-Point

Two-point crossover operates much in the same fashion as one-point crossover. Practically speaking, the primary difference is that end points of the genome are not forced to be the end points of crossover. This type of crossover is appropriate when horizontal genome transcription is acceptable to the problem type.

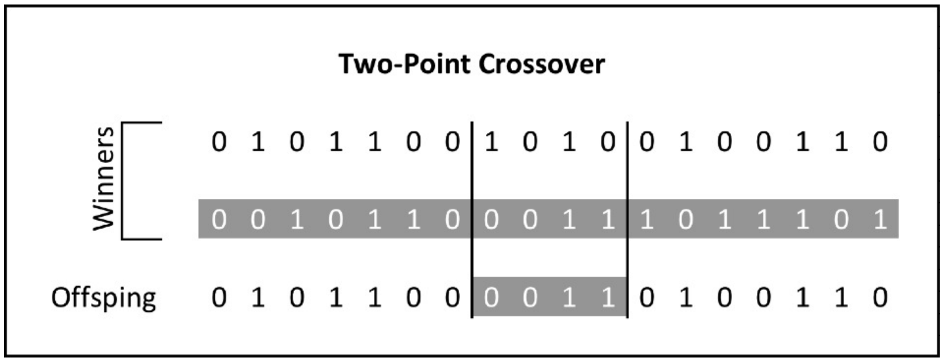

Figure 3: Two-Point Crossover Example with Boolean Population

```
%% C Code Equivalent : Two—Point Crossover
for i = 1:N
    CP1 = round(rand(1)*(G—1)+1);         % Genome Crossover Point 1
    CP2 = round(rand(1)*(G—1)+1);         % Genome Crossover Point 1
    Type = 0;
    if CP1≤CP2
        Type = 1;
    end
    for j = 1:G
        if Type == 1
            if ((j≤CP2) && (j≥CP1))
                idx = (i—1)*2+1;          % Build index from W vector
            else
                idx = (i—1)*2+2;          % Default Index
            end
        else
            if ((j≤CP2) || (j≥CP1))
                idx = (i—1)*2+1;          % Build index from W vector
            else
                idx = (i—1)*2+2;          % Default Index
            end
        end
        Pop2(i,j) = Pop(W(idx),j);        % Add Gene to Genome
    end
end
```

The equivalent C is similar to single point crossover with the exception of some additional condition handling. The same can be said of the efficient approach to two-point crossover in efficient MATLAB code.

```
%% Efficient: Two—Point Crossover
Pop2 = Pop(W(1:2:end),:);                 % Set Pop2 = Pop Winners 1
P2A  = Pop(W(2:2:end),:);                 % Assemble Pop2 Winners 2
Ref  = ones(N,1)*(1:G);                   % Reference Matrix
CP   = sort(round(rand(N,2)*(G—1)+1),2);% Crossover Points
idx  = CP(:,1)*ones(1,G)<Ref & CP(:,2)*ones(1,G)>Ref; % Logical Index
Pop2(idx) = P2A(idx);                     % Recombine Winners
```

Indexing random integers representing genome locations against their column indexes again proves an efficient approach. In this example a second condition in Line 6 is applied to upper bound the crossover. By sorting the randomly generated column indexes in Line 5, additional conditional testing to determine the order of bounds is unnecessary.

4.4. Single-Point Preservation

Due to the recursive nature of more common permutation crossover technique such as Partially Matched Crossover (PMX) and Order Crossover (OX), a single point recombination technique for permutation problems demonstrated. Random permutations can not be crossed unilaterally without potentially producing incomplete genomes. By performing a single location swap, and then 'correcting' any damage to the genome we ensure that it remains a valid permutation.

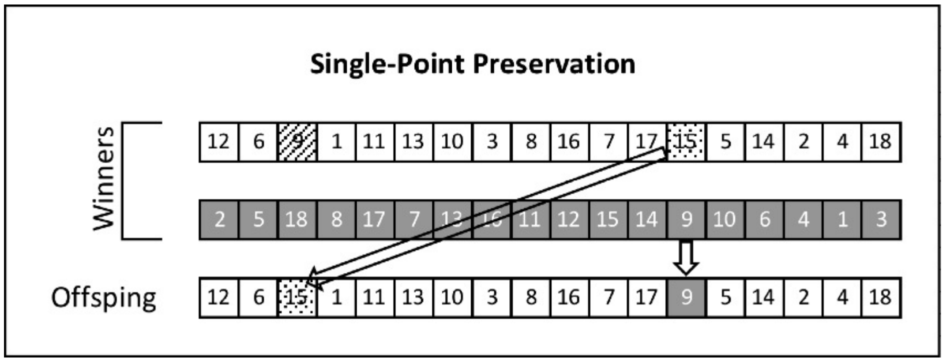

Figure 4: SPP Crossover Example with Permutation Population

```
%% C Code Equivalent: Single—Point Preservation
for i = 1:N
    CP  = round(rand(1)*(G—1)+1);        % Crossover Point
    idx = (i—1)*2+1;                     % First Winner
    OverWritten = Pop(W(idx), CP);       % Value to Overwrite
    OverWriter  = Pop(W(idx+1),CP);      % New Value
    for j = 1:G
        if j == CP
            idx = (i—1)*2+2;
        else
            idx = (i—1)*2+1;
        end
        Pop2(i,j) = Pop(W(idx),j);       % Write Value to Pop2
        if idx—(i—1)*2 == 1 && Pop2(i,j) == OverWriter
            Pop2(i,j) = OverWritten;     % 'Fix' Genome Breaks
        end
    end
end
```

As described in figure 4, a single value is swapped from winner 2 to winner 1. Once the swap is made, the location on winner 1 matching the value of winner 2 at the crossover point is replaced with the original value of winner 1 at the crossover point.

```
1  %% Efficient Single—Point Preservation
2  Pop2 = Pop(W(1:2:end),:);                  % Assemble Pop2 Winners 1
3  P2A  = Pop(W(2:2:end),:);                  % Assemble Pop2 Winners 2
4  Lidx  = sub2ind(size(Pop),[1:N]',round(rand(N,1)*(G—1)+1)); % Select Point
5  vLidx = P2A(Lidx)*ones(1,G);               % Value of Point in Winners 2
6  [r,c] = find(Pop2 == vLidx);               % Location of Values in Winners 1
7  [¬,Ord] = sort(r);                         % Sort Linear Indices
8  r = r(Ord); c = c(Ord);                    % Re—order Linear Indices
9  Lidx2 = sub2ind(size(Pop),r,c);            % Convert to Single Index
10 Pop2(Lidx2) = Pop2(Lidx);                  % Crossover Part 1
11 Pop2(Lidx) = P2A(Lidx);                    % Validate Genomes
```

This the first example which can not be dealt with using a single indexing matrix. In Line 4 a point of crossover for each row is selected. Line 5 retrieves the value of each winner 2 at the crossover point from Line 4. In Line 6 the row and column indices for all crossover points are given for locations in winners 1 with value equal to crossover values from winners 2. Since `find()` natively sorts outputs in column order, Line 7 re-sorts in row order. This row order is used to sort the rows and columns appropriately. In Line 9 the row and column indices are converted to single value indices for ease of use. Finally in Lines 10 to 11 crossover is performed and genomes are repaired.

The demonstrated crossover technique was chosen because it can be implemented efficiently without any iterators. While not widely verified, performing this operation multiple times on adjacent genome locations should provide additional genome recombination effectively.

5. Mutation

To maintain diversity in the population's genomes, a single example of a mutation operator for each of the following population types is presented:

1. Boolean
2. Integer
3. Permutation

5.1. Boolean

For a matrix of 1s and 0s, mutation is applied individually to each element. A user defined 'probability of mutation' per gene is converted to a logical index. The resulting selected values are 'flipped' such that 1s become 0s and 0s become 1s.

```
%% C Code Equivalent: Boolean Mutation
PerMut = 0.01;                           % Prob of Each Element Mutating
for i = 1:N
    for j = 1:G
        idx = rand(1)<PerMut;            % Index for Mutation
        if idx == 1
            Pop2(i,j) = Pop2(i,j)*—1+1; % Flip Bit
        end
    end
```

```
end
```

```
%% Efficient: Boolean Mutation
PerMut = 0.01;                              % Prob of Each Element Mutating
idx = rand(size(Pop2))<PerMut;              % Index of Mutations
Pop2(idx) = Pop2(idx)*-1+1;                 % Flip Bits
```

5.2. Integer

Integer mutation can also be applied in element or gene-wise fashion. Leading to the indexing stage, integer mutation is identical to boolean mutation. When seeding new values though, it is not possible to flip the bit as more than 2 possible values may be present. Instead, a vector of new random integers is produced with length equal to the number of mutation points and mapped to the mutation index.

```
%% C Code Equivalent: Integer Mutation
PerMut = 0.01;                              % Prob of Each Element Mutating
for i = 1:N
    for j = 1:G
        idx = rand(1)<PerMut;               % Index for Mutation
        if idx == 1
            Pop2(i,j) = round(rand(1)*(MaxVal-1)+1); % New Value
        end
    end
end
```

```
%% Efficient: Integer Mutation
PerMut = 0.01;                              % Prob of Each Element Mutating
idx = rand(size(Pop2))<PerMut;              % Index of Mutations
Pop2(idx) = round(rand([1,sum(sum(idx))])*(MaxVal-1)+1); % Mutated Value
```

Note the use of `MaxVal` to seed new genes. The maximum gene value can not be extracted manually (e.g. `max(max(Pop2))`) in case gene diversity loss has eliminated the maximum acceptable gene value. A such, `MaxVal` must be retained from the initial population construction.

5.3. Permutation

Permutation mutations can not be described efficiently on an gene basis with a single index. The variable `PerMut` in this case is the probability of mutation for each genome, not gene.

```
%% C Code Equivalent: Permutation Mutation
PerMut = 0.01;                              % Prob of Each Genome Mutating
for i = 1:N
    idx = rand(1)<PerMut;                   % Index for Mutation
    if idx == 1
```

```
        Loc1 = round(rand(1)*(G-1)+1); % Swap Location 1
        Loc2 = round(rand(1)*(G-1)+1); % Swap Location 2
        Hold = Pop(i,Loc1);                 % Hold Value 1
        Pop2(i,Loc1) = Pop2(i,Loc2);        % Value 1 = Value 2
        Pop2(i,Loc2) = Hold;                % Value 2 = Holder
    end
end
```

To vectorize this code, initially an array of length `N` is created according to `PerMut` to determine which genomes will mutate. Following, a linear index of two locations to 'swap' is created in `Loc1` and `Loc2`. Last, the function `deal()` is used to make the swap in `Loc1` and `Loc2`.

```
%% Efficient: Permutation Mutation
PerMut = 0.5;                                 % Prob of Each Individual Mutating
idx = rand(N,1)<PerMut;                       % Individuals to Mutate
Loc1 = sub2ind(size(Pop2),1:N,round(rand(1,N)*(G-1)+1)); % Index Swap 1
Loc2 = sub2ind(size(Pop2),1:N,round(rand(1,N)*(G-1)+1)); % Index Swap 2
Loc2(idx == 0) = Loc1(idx==0);                % Probabalistically Remove Swaps
[Pop2(Loc1),Pop2(Loc2)] = deal(Pop2(Loc2), Pop2(Loc1)); % Perform Exchange
```

6. Complete Examples

Here, two efficient and complete genetic algorithm examples are shown using the above code snippets. The first example will apply a random boolean population, tournament selection, two point crossover, and boolean mutation. The fitness function to be optimized is the `abs(diff())` or, absolute value of differences between each gene. The optimal solution to this problem is known to be alternating 1s and 0s.

```
%% Genetic Algorithm: Boolean Example
%
% Solves optimization problem where fitness is equal to the
% summed absolute value of genome differences. The optimal genome contains
% alternating values.
%
% Optimal Genome : [0,1,0,1,.....,1,0] OR [1,0,1,0,.....,0,1]

%% Parameters
N = 1000;                                     % Population Size
G = 30;                                       % Genome Size
PerMut = .01;                                 % Probability of Mutation
S = 2;                                        % Tournament Size
Pop = round(rand(N,G));                       % Create Initial Population

for Gen = 1:100                               % Number of Generations
    %% Fitness
    F = sum(abs(diff(Pop,[],2)),2);               % Measure Fitness

    %% Print Stats
    fprintf('Gen: %d    Mean Fitness: %d     Best Fitness: %d\n', Gen, ...
       round(mean(F)), max(F))

    %% Selection (Tournament)
    T   = round(rand(2*N,S)*(N-1)+1);          % Tournaments
    [¬,idx] = max(F(T),[],2);                  % Index to Determine Winners
    W   = T(sub2ind(size(T),(1:2*N)',idx));    % Winners

    %% Crossover (2-Point)
    Pop2 = Pop(W(1:2:end),:);                  % Set Pop2 = Pop Winners 1
    P2A  = Pop(W(2:2:end),:);                  % Assemble Pop2 Winners 2
    Ref  = ones(N,1)*(1:G);                    % Reference Matrix
    CP   = sort(round(rand(N,2)*(G-1)+1),2);% Crossover Points
    idx  = CP(:,1)*ones(1,G)<Ref & CP(:,2)*ones(1,G)>Ref; % Logical Index
    Pop2(idx) = P2A(idx);                      % Recombine Winners

    %% Mutation (Boolean)
    idx = rand(size(Pop2))<PerMut;             % Index of Mutations
    Pop2(idx) = Pop2(idx)*-1+1;                % Flip Bits

    %% Reset
    Pop = Pop2;
end
[¬,BN] = max(F);
disp('Best Genome: ')
disp(Pop(BN,:))
```

After completing the listed 100 generations the best genome is printed as output to the console window. With current parameterization average execution time per generation is 0.0034 seconds on a quad-core 3.24GHz Intel Core i7 CPU running MATLAB 2012b for Unix operating systems.

The second example is a permutation problem. This algorithm utilizes a permutation population, tournament selection, single-preservation crossover, and permutation mutation. Fitness in this case is defined as the variance of the differences among adjacent genome values. The optimal solution to this fitness function `var(diff(Pop,[],2),[],2)` is equal to either `[5,7,3,9,1,10,2,8,4,6]` or `[6,4,8,2,10,1,9,3,7,5]`, for genome length 10.

```
%% Genetic Algorithm: Permutation Example
%
% Solves optimization problem where fitness is equal to the
% variance of genome differences. The optimal genome contains
%
% Optimal Genome : [5,7,3,9,1,10,2,8,4,6] OR [6,4,8,2,10,1,9,3,7,5]

%% Parameters
N = 1000;                                % Population Size
G = 10;                                  % Genome Size
PerMut = .5;                             % Probability of Mutation
S = 2;                                   % Tournament Size
[¬,Pop] = sort(rand(N,G),2);             % Create Initial Population

```

```
for Gen = 1:100                     % Number of Generations

    %% Fitness
    F = var(diff(Pop,[],2),[],2);           % Measure Fitness

    %% Print Stats
    fprintf('Gen: %d    Mean Fitness: %d     Best Fitness: %d\n', Gen, ...
       round(mean(F)), round(max(F)))

    %% Selection (Tournament)
    T   = round(rand(2*N,S)*(N-1)+1);       % Tournaments
    [¬,idx] = max(F(T),[],2);               % Index to Determine Winners
    W   = T(sub2ind(size(T),(1:2*N)',idx)); % Winners

    %% Crossover (Single-Point Preservation)
    Pop2 = Pop(W(1:2:end),:);               % Assemble Pop2 Winners 1
    P2A  = Pop(W(2:2:end),:);               % Assemble Pop2 Winners 2
    Lidx  = sub2ind(size(Pop),[1:N]',round(rand(N,1)*(G-1)+1)); % ...
       Select Point
    vLidx = P2A(Lidx)*ones(1,G);            % Value of Point in Winners 2
    [r,c] = find(Pop2 == vLidx);            % Location of Values in ...
       Winners 1
    [¬,Ord] = sort(r);                      % Sort Linear Indices
    r = r(Ord); c = c(Ord);                 % Re-order Linear Indices
    Lidx2 = sub2ind(size(Pop),r,c);         % Convert to Single Index
    Pop2(Lidx2) = Pop2(Lidx);               % Crossover Part 1
    Pop2(Lidx) = P2A(Lidx);                 % Validate Genomes

    %% Mutation (Permutation)
    idx = rand(N,1)<PerMut;                 % Individuals to Mutate
    Loc1 = sub2ind(size(Pop2),1:N,round(rand(1,N)*(G-1)+1)); % Index ...
       Swap 1
    Loc2 = sub2ind(size(Pop2),1:N,round(rand(1,N)*(G-1)+1)); % Index ...
       Swap 2
    Loc2(idx == 0) = Loc1(idx==0);          % Probabalistically Remove ...
       Swaps
    [Pop2(Loc1),Pop2(Loc2)] = deal(Pop2(Loc2), Pop2(Loc1)); % Perform ...
       Exchange

    %% Reset Population
    Pop = Pop2;
end
[¬,BN] = max(F);                        % Find Best Genome
disp('Best Genome: ')                   % Write Text to Console
disp(Pop(BN,:))                         % Display Best Genome
```

Using the same hardware configuration and new parameters, each generation of the permutation example GA averages a run time of 0.0024 seconds. The speed increase relative to the boolean example is due to the shorter genome. All else being equal, the recombination and mutation procedures for combinatorial problem are less computationally intensive.

Either of these example genetic algorithms can be adapted by altering the fitness function section. For population sizes up to fifty thousand the provided all provided code scales near linearly. Beyond this mark scaling can become problematic. For extremely large population sizes, additional scalability can be gained from converting index (`idx`) variables to utilize linear indices as opposed to indexing matrices (e.g. optimize for sparse matrix operations).

7. References

[1] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Articial Intelligence*, volume 1, pages 161–163, 1985.

[2] D.E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, 1985.

[3] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[4] MATLAB. *version 8.0.0 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, 2012.