An elementary introduction to Matlab programming for stochastic optimization

Mattias Wahde, David Sandberg, and Ola Benderius

August 24, 2020

Introduction

The aim of this document is to provide an introduction to well-structured Matlab programming in general, as well as programming for stochastic optimization algorithms, in particular. A further aim is to raise the level of programming skill among those students who are not so familiar with programming. For those who *are*, the exercise will be quite simple. However, all students should make sure to follow the steps (without shortcuts!) described below. In particular, it is important that you study carefully not only the functionality of the code provided below, but also its structure. During the course, you will be required to write Matlab code that adheres to a given coding standard (see the document `MatlabCodingStandard.pdf`, available on the course web page). Thus, you should make sure, from the beginning, to write program code according to this standard.

Here, we will implement an elementary evolutionary algorithm to solve a simple function optimization problem. The aim will be to find the maximum of the function

$$f(x,y) = \frac{\mathrm{e}^{-x^2-y^2} + \sqrt{5}\sin^2(yx^2) + 2\cos^2(2x+3y)}{1+x^2+y^2}. \tag{1}$$

This text will only introduce the most basic features of Matlab programming, and you should refer to the technical manuals associated with Matlab for more complete information. You should also note that the code has been optimized for *clarity*, not for speed. It is possible to make use of Matlab's vector handling capabilities to speed up the code. However, in this course, the execution speed of the programs will not be the limiting factor, and we shall therefore concentrate on writing *clear* code, as in this document. Furthermore, in other courses (e.g. Intelligent agents and Humanoid robotics), we will use other (and much faster) programming languages, such as `C#`. Thus, learning how to write clear code, according to a given coding standard, is more important than learning how to optimize Matlab code for speed.

The Matlab code will consist of a main program in a Matlab M–file named `FunctionOptimization.m`. This program will, in turn, make use of several other functions, contained in the files `InitializePopulation.m`, `DecodeChromosome.m`, `EvaluateIndividual.m`, `TournamentSelect.m`, `Cross.m`, and `Mutate.m`.

Initialization

Let us first write a skeleton for the main file, containing the parameter definitions. Start Matlab and open a text editor (either the one in Matlab or an external editor) Then, write the following code[1]:

```
1 populationSize = 30;
2 numberOfGenes = 40;
3 crossoverProbability = 0.8;
4 mutationProbability = 0.025;
5 tournamentSelectionParameter = 0.75;
6 variableRange = 3.0;
7 fitness = zeros(populationSize,1);
8
9 % population = InitializePopulation(populationSize, numberOfGenes);
```

We have not yet written the function `InitializePopulation`, and can therefore not call it. The % sign indicates that a row is a comment: Matlab will ignore all text (on that row) after a % sign.

So far, the program only defines six parameters, namely the population size, the number of genes in the chromosomes, the crossover probability, the mutation probability, the tournament selection parameter (defining the probability of selecting the better individual in a tournament involving two individuals), and the range of the variables. Note the naming practice used, in which the first word in a variable name is written using lowercase letters, and all subsequent words (if any) begin with an uppercase letter. Note also the descriptive nature of the variable names: It is better to give a long, descriptive name than a short, generic one. In other words, the variable name `mutationProbability` is much better than, say, `p`.

Next, create an empty folder named `SimpleEvolutionaryAlgorithm`, and save the file described above as `FunctionOptimization.m`. If you use an external editor, make sure to change the file suffix from (say) `.txt` to `.m`. Now test this simple program by typing its name in Matlab's command window:

```
> FunctionOptimization
```

(The `>` symbol is the Matlab prompt). So far, the program only assigns the variable values.

If you now type e.g. `populationSize` in Matlab's command window, Matlab will respond with the value (30) of the population size variable. The semi-colon (`;`) at the end of each line prevents Matlab from printing the result of the operation carried out on the line in question. As a general rule, only relevant output should be printed, and one should be careful not to clog Matlab's output window with a lot of unnecessary output (see also the coding standard).

[1] In all complete code listings, row numbers are printed for clarity and in order to simplify debugging. Those numbers should of course not be entered with writing code! Code snippets (showing only a part of a function or the main program) are given without line numbers.

Now, open the text editor again. Let us write the function `InitializePopulation`, which assigns random values to all genes, in all chromosomes contained in the population. Type the following, and then save the file (as `InitializePopulation.m`)

```
function population = InitializePopulation(populationSize, nGenes)

  population = zeros(populationSize, nGenes);
  for i = 1:populationSize
    for j = 1:nGenes
        s = rand;
        if (s < 0.5)
            population(i,j)=0;
        else
            population(i,j)=1;
        end
    end
  end

end
```

This function will generate a population of binary chromosomes. Note that Matlab prefers to work with vectors and matrices, and that its `for` loops are a bit slower. Therefore, the `InitializePopulation` function would be much faster if it were defined as

```
function population = InitializePopulation(populationSize,numberOfGenes);

population = fix(2.0*rand(populationSize,numberOfGenes));
```

where the `fix` function rounds (downwards) to the nearest integer. As mentioned above, for clarity, we will most often use `for` loops in this introduction, rather than the more compact matrix notation. When you write programs to solve the home problems (at least in this course), you should write code optimized for clarity rather than speed. In other courses (for example, Intelligent agents), there will be some time-critical operations (such as, for instance, image processing), where the speed of the code will be crucial. However, in those cases, we will not use Matlab anyway.

It should also be noted that, in the case of temporary variables with limited scope, one may (without violating the coding standard) use a rather short name, e.g. `nGenes` instead of `numberOfGenes`. However, even short variable names should follow the standard for naming variables, i.e. with the first word (or prefix) in lower case, and all subsequent words (if any) beginning with an uppercase letter. Finally, note that the values of local variables (such as e.g. `nGenes`) are lost when Matlab exits the function.

Now, remove the comment (%) in the `FunctionOptimization.m` file, save all files, and run the program by again typing

```
> FunctionOptimization
```

There should be no output. If you type

```
> population
```

the entire population (i.e. all the chromosomes) will be listed. They should contain only 0s and 1s.

As an aside, note that the variables defined when executing the program will reside in Matlab's working memory until cleared (or overwritten, for example by executing the program again). A common error in submitted solutions to home problems is that the working memory may have contained crucial information that is not properly set by the program itself, and therefore will not, perhaps, be available if the program is executed directly after starting Matlab. Thus, before submitting the solution to a home problem, exit Matlab, then restart Matlab and run the program directly, to make sure that it works properly. Also, you can of course add the *clear all* command at the very beginning of your main program.

Evaluation

The next step is to evaluate the individuals of the population. Modify the main program (`FunctionOptimization.m`) to read

```
populationSize = 30;
numberOfGenes = 40;
crossoverProbability = 0.8;
mutationProbability = 0.025;
tournamentSelectionParameter = 0.75;
variableRange = 3.0;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize, numberOfGenes);

for i = 1:populationSize
  chromosome = population(i,:);
  x = DecodeChromosome(chromosome, variableRange);
  fitness(i) = EvaluateIndividual(x);
end
```

Once the two functions `DecodeChromosome` and `EvaluateIndividual` have been written (see below), the added lines will loop through the entire population, extract and decode chromosomes, and evaluate the corresponding individual. Note that the step defining the chromosome is not absolutely necessary: It would be possible to write

```
x = DecodeChromosome(population(i,:),variableRange);
```

However, the definition of a separate chromosome variable makes the code clearer (see also Sect. 3.2 in the coding standard). It is a good idea to develop the habit of

writing easily readable code. Note that the notation `population(i,:)` indicates a vector containing all elements on row `i` of the matrix `population`.

Next, open the text editor, and write the function `DecodeChromosome` (saved as `DecodeChromosome.m`) as follows:

```
function x = DecodeChromosome(chromosome,variableRange)

  nGenes = size(chromosome,2);
  nHalf = fix(nGenes/2);

  x(1) = 0.0;
  for j = 1:nHalf
    x(1) = x(1) + chromosome(j)*2^(-j);
  end
  x(1) = -variableRange + 2*variableRange*x(1)/(1 - 2^(-nHalf));

  x(2) = 0.0;
  for j = 1:nHalf
    x(2) = x(2) + chromosome(j+nHalf)*2^(-j);
  end
  x(2) = -variableRange + 2*variableRange*x(2)/(1 - 2^(-nHalf));

end
```

This function uses the genes in the first half of the chromosomes to obtain a value of `x(1)` in the range [0,1], and the remaining genes to obtain a value of `x(2)` in the same range. `x(1)` and `x(2)` are then rescaled to the interval [`-variableRange`,`variableRange`] (see also Eq. (3.2) in the course book, p. 41).

Note that the `DecodeChromosome` function is by no means general: It assumes that the number of genes is even, and that only two variables are to be generated. However, it is easy to modify this function, so that it can generate any number of variables, defining each variable using a given number of genes. You will be required to write such code in connection with the home problems. Now write the file `EvaluateIndividual.m` as follows:

```
function f = EvaluateIndividual(x)

    fNumerator1 = exp(-x(1)^2-x(2)^2);
    fNumerator2 = sqrt(5)*(sin(x(2)*x(1)*x(1))^2);
    fNumerator3 = 2*(cos(2*x(1) + 3*x(2))^2);

    fDenominator = 1 + x(1)^2 + x(2)^2;

    f = (fNumerator1 + fNumerator2 + fNumerator3)/fDenominator;

end
```

`EvaluateIndividual` encodes the function $f(x, y)$ given in Eq. (1). In keeping with the coding standard, it does so by dividing the rather complex expression into smaller bits, which can more easily be debugged to find potential errors.

Since, in this case, the task is to maximize the function, we simply use the value of the function as fitness: Higher function values also mean higher *fitness*. Note: It is common to make mistakes in `EvaluateIndividual` function. Check your code carefully before proceeding!

Now, save all open files, and test the program by typing `FunctionOptimization` in the Matlab command window. If you now type `fitness`, a list of 30 (=`populationSize`) fitness values should appear. Since the population was generated randomly, it is unlikely that any of the generated fitness values are near the optimum (which, so far, is unknown!). The next step is to try to improve the population, through selection, crossover, and mutation.

Selection

For the selection step, we will use tournament selection with tournament size 2. (As an exercise, try also to implement roulette wheel selection!). Again, open a new file with a suitable text editor, and enter the following:

```
function iSelected = TournamentSelect(fitness, pTournament)

  populationSize = size(fitness,1);
  iTmp1 = 1 + fix(rand*populationSize);
  iTmp2 = 1 + fix(rand*populationSize);

  r = rand;

  if (r < pTournament)
    if (fitness(iTmp1) > fitness(iTmp2))
      iSelected = iTmp1;
    else
      iSelected = iTmp2;
    end
  else
    if (fitness(iTmp1) > fitness(iTmp2))
       iSelected = iTmp2;
    else
       iSelected = iTmp1;
    end
  end

end
```

Save the file as `TournamentSelect.m`. This function chooses the (index of the) better of two randomly selected individuals with the probability given by the tournament selection parameter p_{tour} (set to 0.75 above). With probability $1 - p_{\text{tour}}$ the (index of the) worse of the two individuals is selected. Now, modify the main program (`FunctionOptimization.m`) so that it takes the following form

```
populationSize = 30;
numberOfGenes = 40;
crossoverProbability = 0.8;
mutationProbability = 0.025;
tournamentSelectionParameter = 0.75;
variableRange = 3.0;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize, numberOfGenes);

for i = 1:populationSize
  chromosome = population(i,:);
  x = DecodeChromosome(chromosome, variableRange);
  fitness(i) = EvaluateIndividual(x);
end

tempPopulation = population;

for i = 1:2:populationSize
  i1 = TournamentSelect(fitness,tournamentSelectionParameter);
  i2 = TournamentSelect(fitness,tournamentSelectionParameter);
  chromosome1 = population(i1,:);
  chromosome2 = population(i2,:);
  tempPopulation(i,:) = chromosome1;
  tempPopulation(i+1,:) = chromosome2;
end

population = tempPopulation;
```

The current version of the code first generates a random set of chromosomes, which is then decoded and evaluated. Next, a temporary population is generated by means of tournament selection. The notation `1:2:populationSize` in the `for` loop indicates that only every other i value will be considered, i.e. `i` $= 1, 3, 5, 7, \ldots$. Finally, the temporary population replaces the original population.

Crossover and mutation

So far, the selected individuals are copied unchanged to the next generation (i.e. to the new population). Of course, in order to improve the results, one should modify the selected individuals as well. In genetic algorithms, two operators are normally used for modifying individuals: crossover and mutation. As the next step, enter the following code in a suitable text editor

```
function newChromosomePair = Cross(chromosome1,chromosome2)

  nGenes = size(chromosome1,2); % Both chromosomes must have
                                % the same length!
  crossoverPoint = 1 + fix(rand*(nGenes-1));

  newChromosomePair = zeros(2,nGenes);
  for j = 1:nGenes
    if (j <= crossoverPoint)
      newChromosomePair(1,j) = chromosome1(j);
      newChromosomePair(2,j) = chromosome2(j);
    else
      newChromosomePair(1,j) = chromosome2(j);
      newChromosomePair(2,j) = chromosome1(j);
    end
  end

end
```

and save it as `Cross.m`. This function defines a crossover point randomly between two genes in the chromosomes, and makes two new temporary chromosomes using one–point crossover. Now, in general, crossover is a very efficient operator which sometimes can cause the population to get stuck in a local optimum, so called *premature convergence*; see pp. 66-70 in the course book. Therefore, it is common to apply crossover only with a certain probability; see the code listing for the main program on the next page.

Next, in order to implement mutations, enter the following code

```
function mutatedChromosome = Mutate(chromosome,mutationProbability)

  nGenes = size(chromosome,2);
  mutatedChromosome = chromosome;
  for j = 1:nGenes
    r = rand;
    if (r < mutationProbability)
      mutatedChromosome(j) = 1-chromosome(j);
    end
  end

end
```

and save it as `Mutate.m`. This function loops through all the genes in a chromosome, and flips the corresponding gene (bit) with probability p_{mut}, as specified by the `mutationProbability` variable. The implementation is specifically written for binary chromosomes. For chromosomes in which the genes are floating-point numbers (typically in the range 0 to 1), one would obtain the new (mutated) value simply by calling the `rand` function.

Now we can add crossover and mutation to the main program, which then takes the following form:

```
populationSize = 30;
numberOfGenes = 40;
crossoverProbability = 0.8;
mutationProbability = 0.025;
tournamentSelectionParameter = 0.75;
variableRange = 3.0;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize, numberOfGenes);

for i = 1:populationSize
  chromosome = population(i,:);
  x = DecodeChromosome(chromosome, variableRange);
  fitness(i) = EvaluateIndividual(x);
end

tempPopulation = population;

for i = 1:2:populationSize
  i1 = TournamentSelect(fitness,tournamentSelectionParameter);
  i2 = TournamentSelect(fitness,tournamentSelectionParameter);
  chromosome1 = population(i1,:);
  chromosome2 = population(i2,:);

  r = rand;
  if (r < crossoverProbability)
    newChromosomePair = Cross(chromosome1,chromosome2);
    tempPopulation(i,:) = newChromosomePair(1,:);
    tempPopulation(i+1,:) = newChromosomePair(2,:);
  else
    tempPopulation(i,:) = chromosome1;
    tempPopulation(i+1,:) = chromosome2;
  end
end % Loop over population

for i = 1:populationSize
  originalChromosome = tempPopulation(i,:);
  mutatedChromosome = Mutate(originalChromosome,mutationProbability);
  tempPopulation(i,:) = mutatedChromosome;
end

population = tempPopulation;
```

Here, as noted above, crossover is applied only with a certain probability (the `crossoverProbability`). Thus, after selection and crossover, some new individuals will have been formed via crossover, whereas others will, so far, be unchanged. After these two steps, all individuals are subjected to mutations (but see the next section for an exception!)

Elitism

In order to achieve a monotonous increase in the fitness values, we should use elitism. To add this feature, modify the evaluation part of `FunctionOptimization.m` to read

```
maximumFitness = 0.0; % Assumes non-negative fitness values!
xBest = zeros(1,2); % [0 0]
bestIndividualIndex = 0;
for i = 1:populationSize
  chromosome = population(i,:);
  x = DecodeChromosome(chromosome, variableRange);
  fitness(i) = EvaluateIndividual(x);
  if (fitness(i) > maximumFitness)
     maximumFitness = fitness(i);
     bestIndividualIndex = i;
     xBest = x;
  end
end
```

Next (important!), add the following line just before the line that overwrites the old population:

```
tempPopulation(1,:) = population(bestIndividualIndex,:);
```

With these changes (see also the complete listing below), the program will keep track of the best individual, and insert a copy of this individual at the (arbitrarily chosen) first position in the new population.

Complete program

So far, the program just evaluates a single generation and then generates a new population without evaluating it. Almost always, one needs to iterate the processes of evaluation, selection, and reproduction many times. Thus, define a variable `numberOfGenerations` just after the definition of the variable range in the main program (`FunctionOptimization.m`), and set its value to 100.

Next, add a `for` loop that iterates over the number of generations just defined. Finally, add a line that prints the best fitness value, and the corresponding coordinates (`x`), at the end of each generation. The printouts (after each generation) will be removed later, and will be replaced by graphical output. However, the final printout (see the last lines below) should be kept. The main program then takes the form: (note the *clear all* command that has been added on the first line)

```
1 clear all;
2
3 populationSize = 30;
4 numberOfGenes = 40;
5 crossoverProbability = 0.8;
```

```
mutationProbability = 0.025;
tournamentSelectionParameter = 0.75;
variableRange = 3.0;
numberOfGenerations = 100;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize, numberOfGenes);

for iGeneration = 1:numberOfGenerations

  maximumFitness = 0.0; % Assumes non-negative fitness values!
  xBest = zeros(1,2); % [0 0]
  bestIndividualIndex = 0;
  for i = 1:populationSize
    chromosome = population(i,:);
    x = DecodeChromosome(chromosome, variableRange);
    fitness(i) = EvaluateIndividual(x);
    if (fitness(i) > maximumFitness)
     maximumFitness = fitness(i);
     bestIndividualIndex = i;
     xBest = x;
    end
  end

  % Printout
  disp('xBest');
  disp(xBest);
  disp('maximumFitness');
  disp(maximumFitness);

  tempPopulation = population;

  for i = 1:2:populationSize
    i1 = TournamentSelect(fitness,tournamentSelectionParameter);
    i2 = TournamentSelect(fitness,tournamentSelectionParameter);
    chromosome1 = population(i1,:);
    chromosome2 = population(i2,:);

    r = rand;
    if (r < crossoverProbability)
      newChromosomePair = Cross(chromosome1,chromosome2);
      tempPopulation(i,:) = newChromosomePair(1,:);
      tempPopulation(i+1,:) = newChromosomePair(2,:);
    else
      tempPopulation(i,:) = chromosome1;
      tempPopulation(i+1,:) = chromosome2;
    end
  end % Loop over population

  for i = 1:populationSize
    originalChromosome = tempPopulation(i,:);
    mutatedChromosome = Mutate(originalChromosome,mutationProbability);
    tempPopulation(i,:) = mutatedChromosome;
  end
```

```
60
61    tempPopulation(1,:) = population(bestIndividualIndex,:);
62    population = tempPopulation;
63
64  end % Loop over generations
65
66  % Print final result
67  disp('xBest');
68  disp(xBest);
69  disp('maximumFitness');
70  disp(maximumFitness);
```

Note the indentation, which increases the readability of the code. Note also the last lines, which print the maximum fitness value found as well as the corresponding variable values *at the end* of the run. In fact, the program prints this information for each generation which, as indicated above, is perhaps a bit unnecessary. The corresponding lines can be removed, once a graphical output (see below) is available.

Using graphics

You have now learned how to write a basic EA in Matlab. Obviously, many refinements can be made. For instance, a nice graphical user interface (GUI) makes it easier to study the progress of the EA. As an example, with the code additions described below, Matlab will plot the progress (the best fitness in each generation) of the EA in a window and also write the value of `maxFitness` in the same window.

First, a new figure needs to be created and initialized with appropriate parameters; the following lines of code do just that:

```
fitnessFigureHandle = figure;
hold on;
set(fitnessFigureHandle, 'Position', [50,50,500,200]);
set(fitnessFigureHandle, 'DoubleBuffer','on');
axis([1 numberOfGenerations 2.5 3]);
bestPlotHandle = plot(1:numberOfGenerations,zeros(1,numberOfGenerations));
textHandle = text(30,2.6,sprintf('best: %4.3f',0.0));
hold off;
drawnow;
```

These lines of code should be placed in the `FunctionOptimization.m` file, after the initialization of variables but before the line:

```
population = InitializePopulation(populationSize, numberOfGenes);
```

Please consult the Matlab documentation for the commands and parameters used in the above code snippet. In particular, you may wish to study the concept of *Handle graphics*.

Here (and in the surface graphics below), for simplicity, some numerical constants (e.g. the plot range, the position of the plot window etc.) have been entered as

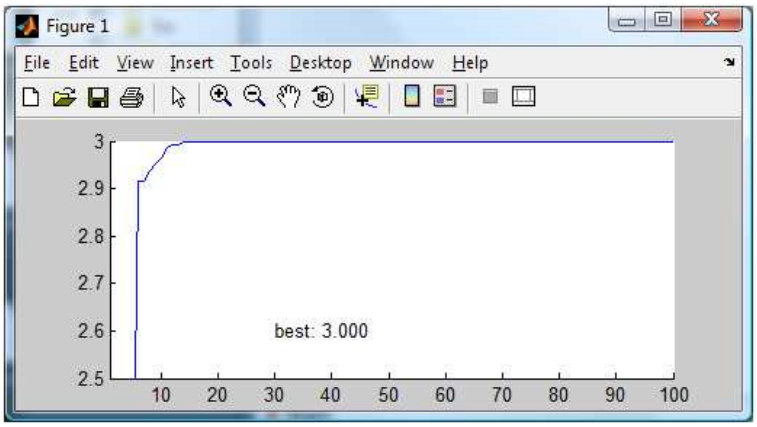

Figure 1: The plot window showing the performance of the GA.

numerical values at the point where they are used. In order to follow the coding standard strictly, one should instead use variable names also for these parameters (e.g. `plotWindowLeft`, `plotWindowTop` etc.), and place the corresponding variables at the beginning of the code.

After having created a figure and its handle `fitnessFigureHandle`, the following lines of code are used to update a vector containing the highest fitness value for each generation and plot that vector in the figure maintained by the `fitnessFigureHandle` variable. These lines of code should be placed *before* the `end` of the `for iGeneration = 1:numberOfGenerations` loop, i.e. just above the last `end` in the `FunctionOptimization.m` file.

```
plotvector = get(bestPlotHandle,'YData');
plotvector(iGeneration) = maximumFitness;
set(bestPlotHandle,'YData',plotvector);
set(textHandle,'String',sprintf('best: %4.3f',maximumFitness));
drawnow;
```

Now run the program and notice how the progress of the EA is plotted in the figure, as shown in Fig. 1.

Plotting the solution surface

As the function under consideration is a real-valued function of two variables, it is possible to plot the function surface using 3D graphics in Matlab. As with the initialization of the figure for plotting the progress of the highest fitness, the following lines of code, which initialize the figure for the surface plot, should be placed after the initialization of the variables but before the line:

```
population = InitializePopulation(populationSize, numberOfGenes);
```

in the `FunctionOptimization.m` file:

```
surfaceFigureHandle = figure;
hold on;
set(surfaceFigureHandle, 'DoubleBuffer','on');
delta = 0.1;
limit = fix(2*variableRange/delta) + 1;
[xValues,yValues] = meshgrid(-variableRange:delta:variableRange,...
    -variableRange:delta:variableRange);
zValues = zeros(limit,limit);
for j = 1:limit
    for k = 1:limit
        zValues(j,k) = EvaluateIndividual([xValues(j,k) yValues(j,k)]);
    end
end
surfl(xValues,yValues,zValues)
colormap gray;
shading interp;
view([-7 -9 10]);
decodedPopulation = zeros(populationSize,2);
populationPlotHandle = plot3(decodedPopulation(:,1), ...
    decodedPopulation(:,2),fitness(:),'kp');
hold off;
drawnow;
```

In addition, the above code snippet also creates a handle to a 3D plot which contains the decoded chromosome and fitness of each individual in the population (i.e. the `x` and `y` values with the corresponding function value). We will use this handle, `populationPlotHandle`, to update and re-plot the individuals in each generation, in the surface plot figure. Note that long expressions can be split over several lines, using `...`.

Now, the matrix `decodedPopulation` must be updated to contain the decoded `x` and `y` values of each individual. So, change the code of `FunctionOptimization.m` to include such a statement (the second line from the bottom in the following code snippet) as follows:

```
for iGeneration = 1:numberOfGenerations
    maximumFitness = 0.0; % Assumes non-negative fitness values!
    for i = 1:populationSize
        chromosome = population(i,:);
        x = DecodeChromosome(chromosome, variableRange);
        decodedPopulation(i,:) = x; % The new statement
        fitness(i) = EvaluateIndividual(x);

   ... (etc.)
```

Finally, a statement to update the plot of the population is needed. Change the last lines of code in the `FunctionOptimization.m` file to:

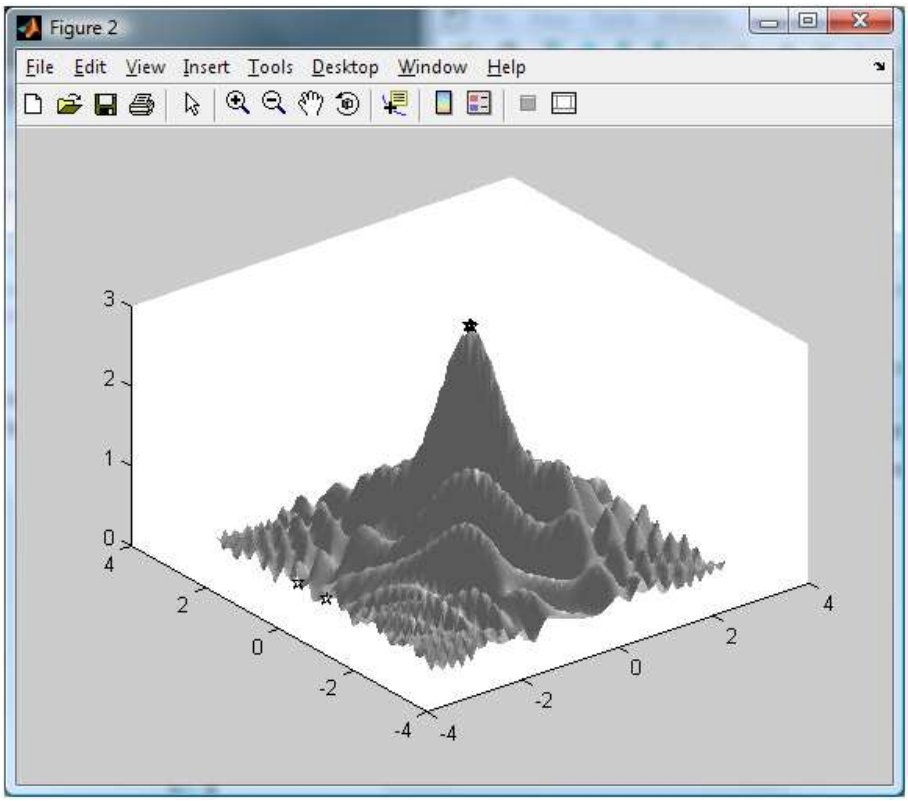

Figure 2: The plot window showing a three-dimensional view of the population.

```
    set(bestPlotHandle,'YData',plotvector);
    set(textHandle,'String',sprintf('best: %4.3f',maximumFitness));
    % The following statement is new
    set(populationPlotHandle,'XData',decodedPopulation(:,1),'YData', ...
        decodedPopulation(:,2),'ZData',fitness(:));
    drawnow;
end
```

The entire code listing is given below. Now run the program and observe the surface plot (shown in Fig. 2) as well as the symbols representing the individuals on this surface.

Final code listing

```
1 clear all;
2
3 populationSize = 30;
4 numberOfGenes = 40;
5 crossoverProbability = 0.8;
6 mutationProbability = 0.025;
```

```
tournamentSelectionParameter = 0.75;
variableRange = 3.0;
numberOfGenerations = 100;
fitness = zeros(populationSize,1);

fitnessFigureHandle = figure;
hold on;
set(fitnessFigureHandle, 'Position', [50,50,500,200]);
set(fitnessFigureHandle, 'DoubleBuffer','on');
axis([1 numberOfGenerations 2.5 3]);
bestPlotHandle = plot(1:numberOfGenerations,...
                 zeros(1,numberOfGenerations));
textHandle = text(30,2.6,sprintf('best: %4.3f',0.0));
hold off;
drawnow;

surfaceFigureHandle = figure;
hold on;
set(surfaceFigureHandle, 'DoubleBuffer','on');
delta = 0.1;
limit = fix(2*variableRange/delta) + 1;
[xValues,yValues] = meshgrid(-variableRange:delta:variableRange,...
    -variableRange:delta:variableRange);
zValues = zeros(limit,limit);
for j = 1:limit
    for k = 1:limit
        zValues(j,k) = ...
        EvaluateIndividual([xValues(j,k) yValues(j,k)]);
    end
end
surfl(xValues,yValues,zValues)
colormap gray;
shading interp;
view([-7 -9 10]);
decodedPopulation = zeros(populationSize,2);
populationPlotHandle = plot3(decodedPopulation(:,1), ...
    decodedPopulation(:,2),fitness(:),'kp');
hold off;
drawnow;

population = InitializePopulation(populationSize, numberOfGenes);

for iGeneration = 1:numberOfGenerations
    maximumFitness = 0.0; % Assumes non-negative fitness values!
    xBest = zeros(1,2); % [0 0]
    bestIndividualIndex = 0;
    for i = 1:populationSize
        chromosome = population(i,:);
        x = DecodeChromosome(chromosome, variableRange);
        decodedPopulation(i,:) = x;
        fitness(i) = EvaluateIndividual(x);
        if (fitness(i) > maximumFitness)
            maximumFitness = fitness(i);
            bestIndividualIndex = i;
```

```
            xBest = x;
        end
    end

%  disp('xBest'); % Output suppressed. Retained for debugging.
%  disp(xBest);
%  disp('maximumFitness');
%  disp(maximumFitness);


    tempPopulation = population;

    for i = 1:2:populationSize
        i1 = TournamentSelect(fitness,tournamentSelectionParameter);
        i2 = TournamentSelect(fitness,tournamentSelectionParameter);
        chromosome1 = population(i1,:);
        chromosome2 = population(i2,:);

        r = rand;
        if (r < crossoverProbability)
            newChromosomePair = Cross(chromosome1,chromosome2);
            tempPopulation(i,:) = newChromosomePair(1,:);
            tempPopulation(i+1,:) = newChromosomePair(2,:);
        else
            tempPopulation(i,:) = chromosome1;
            tempPopulation(i+1,:) = chromosome2;
        end
    end

    for i = 1:populationSize
        originalChromosome = tempPopulation(i,:);
        mutatedChromosome = Mutate(originalChromosome,...
                                   mutationProbability);
        tempPopulation(i,:) = mutatedChromosome;
    end

    tempPopulation(1,:) = population(bestIndividualIndex,:);
    population = tempPopulation;

    plotvector = get(bestPlotHandle,'YData');
    plotvector(iGeneration) = maximumFitness;
    set(bestPlotHandle,'YData',plotvector);
    set(textHandle,'String',sprintf('best: %4.3f',maximumFitness));
    set(populationPlotHandle,'XData',decodedPopulation(:,1),...
       'YData', decodedPopulation(:,2),'ZData',fitness(:));
    drawnow;
end

format long;
disp('xBest');
disp(xBest);
disp('maximumFitness');
disp(maximumFitness);
```