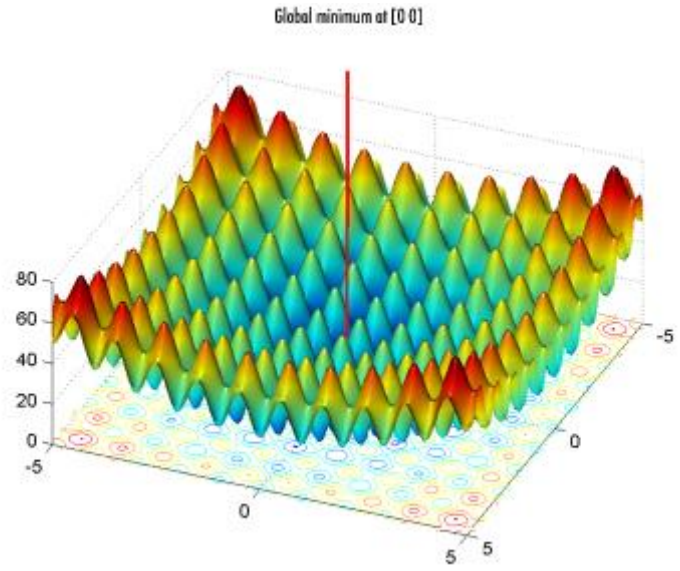
GENETIC ALGORITHMS/EVOLUTION STRATEGIES LAB class guide

Task 1- Rastrigin's Function – Using GA App

PROBLEM DESCRIPTION: Find the minimum of Rastrigin's function, a function that is often used to test the genetic algorithm.

For two independent variables, Rastrigin's function is defined as

$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$



As the plot shows, Rastrigin's function has many local minima—the "valleys" in the plot. However, the function has just one global minimum, which occurs at the point [0 0] in the *x-y* plane, as indicated by the vertical line in the plot, where the value of the function is 0. At any local minimum other than [0 0], the value of Rastrigin's function is greater than 0. The farther the local minimum is from the origin, the larger the value of the function is at that point.

Finding the Minimum from the App

Global Optimization Toolbox software contains the rastriginsfcn.m file, which computes the values of Rastrigin's function. The following statement plots a Rastrigin's function.

```
plotobjective(@rastriginsfcn,[-5 5; -5 5]);
```

Let's find the minimum. Do the following steps:

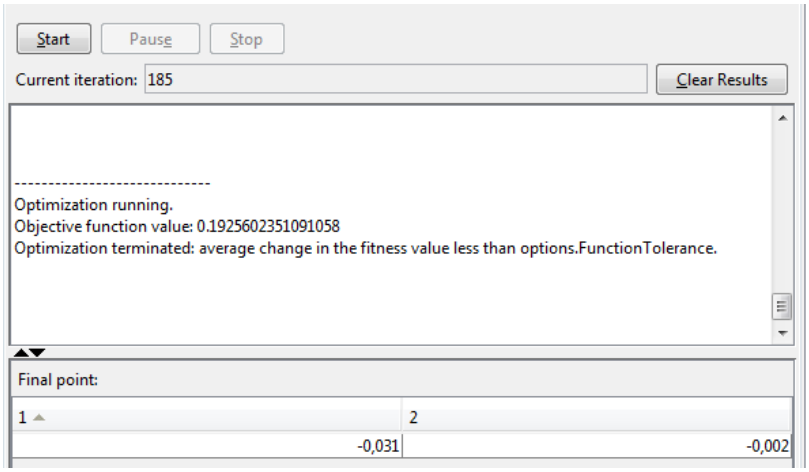
1.Enter optimtool('ga') at the command line to open the Optimization app.

(or optimtool and then select solver ga; or go to APPS and select Optimization icon and then solver ga.)

2.Enter the following in the Optimization app:

- In the **Fitness function** field, enter @rastriginsfcn.
- In the **Number of variables** field, enter 2, the number of independent variables for Rastrigin's function.

3. Click the **Start** button in the **Run solver and view results**.
4. When the algorithm is finished, the **Run solver and view results** pane appears as shown in the following figure. Your numerical results might differ from those in the figure, since genetic algoritm is stochastic.



The display shows:

- The final value of the fitness function when the algorithm terminated:

  Objective function value: 0.1925602351091058
  Note that the value shown is very close to the actual minimum value of Rastrigin's function, which is 0.

- The reason the algorithm terminated.
  Optimization terminated: average change in the fitness value less than options.FunctionTolerance.
  This means that the algorithm stopped because the average change in the fitness function value over Stall generations is less than the Function tolerance. By default, Stall generations is set to 50 and Function tolerance is set to 1e-6.

- The final point, which in this example is [-0,031 -0.002].

Note: You can find a quick reference of all the options available for each parameter in the top right hand side of the optimization app, pushing the >> symbol.

Finding the Minimum from the Command Line

To find the minimum of Rastrigin's function from the command line, enter

```
[x fval exitflag] = ga(@rastriginsfcn, 2)
```

This returns:

```
Optimization terminated:
average change in the fitness value less than options.TolFun.

x =
   0.0071    0.0220
fval =
   0.1058
exitflag =
    1
```

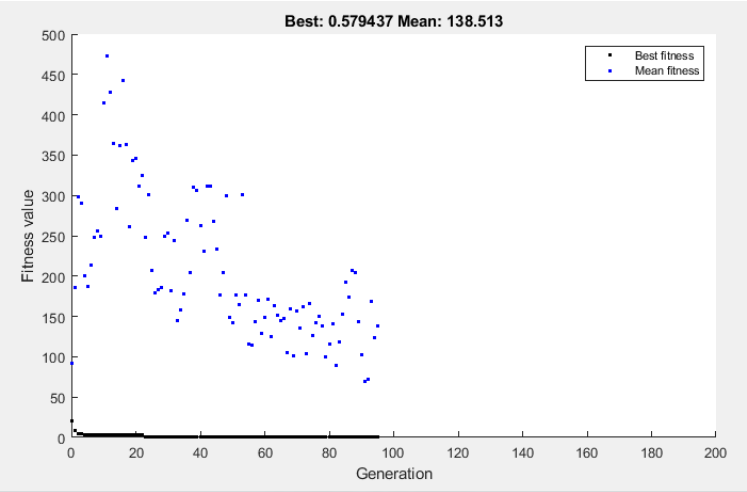
- x is the final point returned by the algorithm.
- fval is the fitness function value at the final point.
- exitflag is integer value corresponding to the reason that the algorithm terminated.

1 Average change in value of the fitness function over options.StallGenLimit generations less than options.TolFun
3 The value of the fitness function did not change in options.StallGenLimit generations
4 Magnitude of step smaller than machine precision
5 Fitness limit reached
0 Maximum number of generations exceeded.

Displaying Plots

The **Plot functions** pane (right hand side of the App) enables you to display various plots that provide information about the genetic algorithm while it is running. This information can help you change options to improve the performance of the algorithm. For example, to plot the best and mean values of the fitness function at each generation, select **Best fitness**.

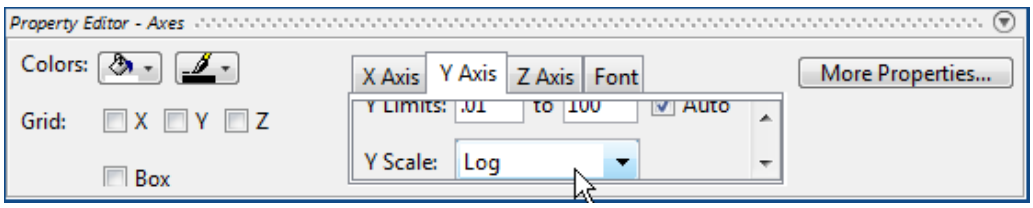
When you click **Start**, the Optimization app displays a plot of the best and mean values of the fitness function at each generation.



The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The plot also displays the best and mean values in the current generation numerically at the top. Since the algorithm cannot improve the best fitness value, i.e. the average change in the fitness function value over 50 generations is less than the 1e-6, the algorithm stops

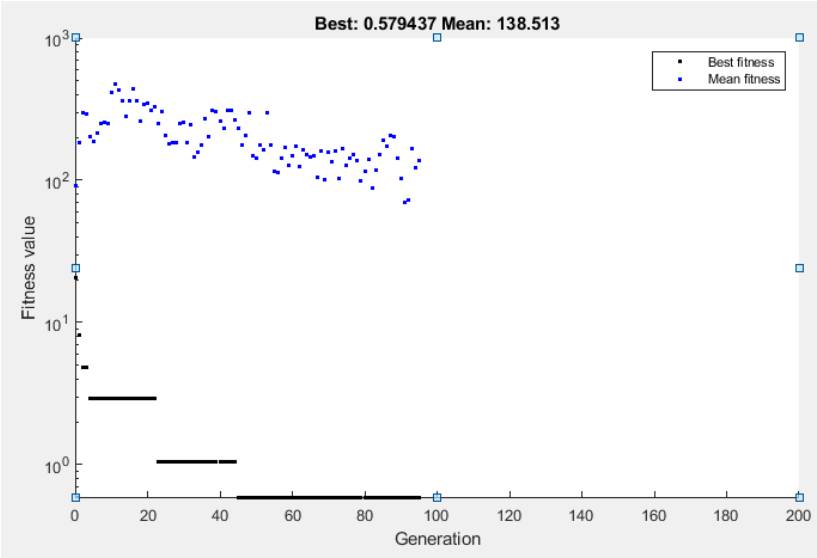
To get a better picture of how much the best fitness values are decreasing, you can change the scaling of the *y*-axis in the plot to logarithmic scaling. To do so,

1. Select **Property Editor** from the **view** menu in the plot window to open the Property Editor attached to your figure window as shown below.



2. Click the **Y Axis** tab.
3. In the **Y Scale** pane, select **Log**.

   The plot now appears as shown in the following figure.



Typically, the best fitness value improves rapidly in the early generations, when the individuals are farther from the optimum. The best fitness value improves more slowly in later generations, whose populations are closer to the optimal point.

Task 2- Determine the Range of the initial population (Rastrigin's Function)

One of the most important factors that determines the performance of the genetic algorithm is the *diversity* of the population. If the average distance between individuals is large, the diversity is high; if the average distance is small, the diversity is low. If the diversity is too high or too low, the genetic algorithm might not perform well.

Setting the Initial Range

By default, **ga** creates a random initial population using a creation function. Different creation functions can be selected, i.e. Constraint dependent (default), Uniform, Feasible population.

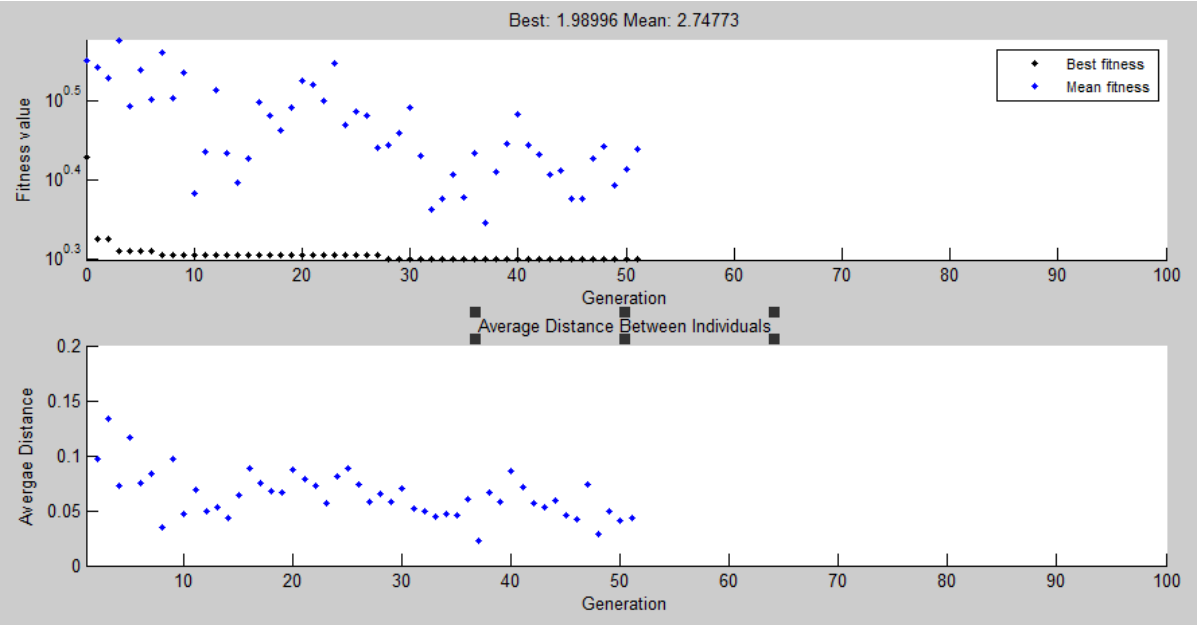
You can specify the range of the vectors in the initial population in the **Initial range** field in **Population** options. The initial range restricts the range of the points in the *initial* population by specifying the lower and upper bounds. Subsequent generations can contain points whose entries do not lie in the initial range.

If you like to set upper and lower bounds for all generations you can do it in the **Bounds** fields in the **Constraints** panel (left hand side of the Optimization Tool window).

If you know approximately where the solution to a problem lies, specify the initial range so that it contains your guess for the solution. However, the genetic algorithm can find the solution even if it does not lie in the initial range, if the population has enough diversity.

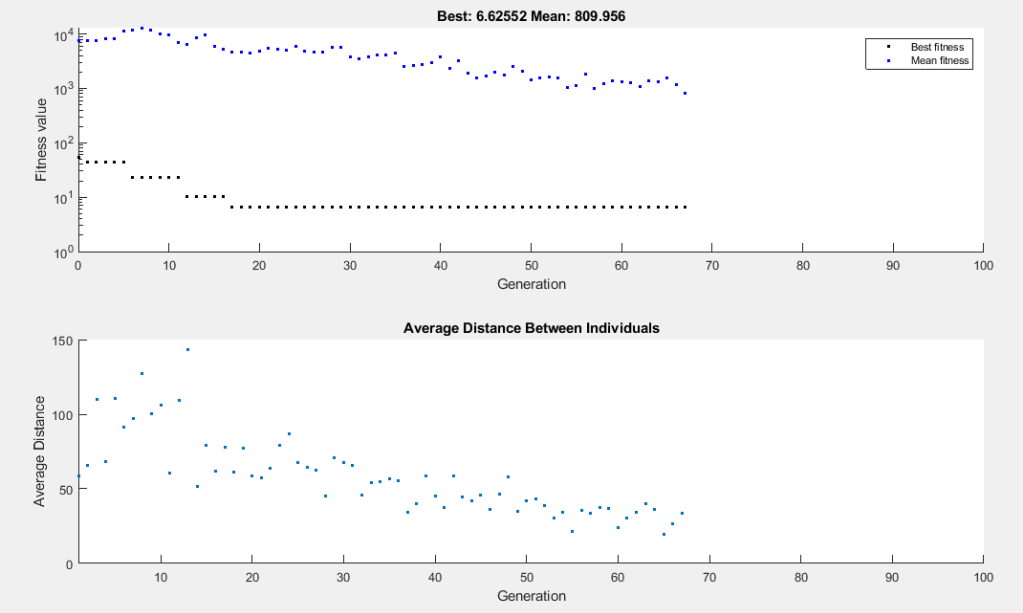
How the initial range affects the performance of the genetic algorithm?

- Set **Initial range** in the **Population** pane of the **Options** pane to [1;1.1].
- Select **Best fitness** and **Distance** in the **Plot functions** pane.
- Specify **Stopping criteria / Generations** to 100.
- Click **Start** in **Run solver and view results**. Although the results of genetic algorithm computations are random, your results are similar to the following figure, with a best fitness function value of approximately 2.



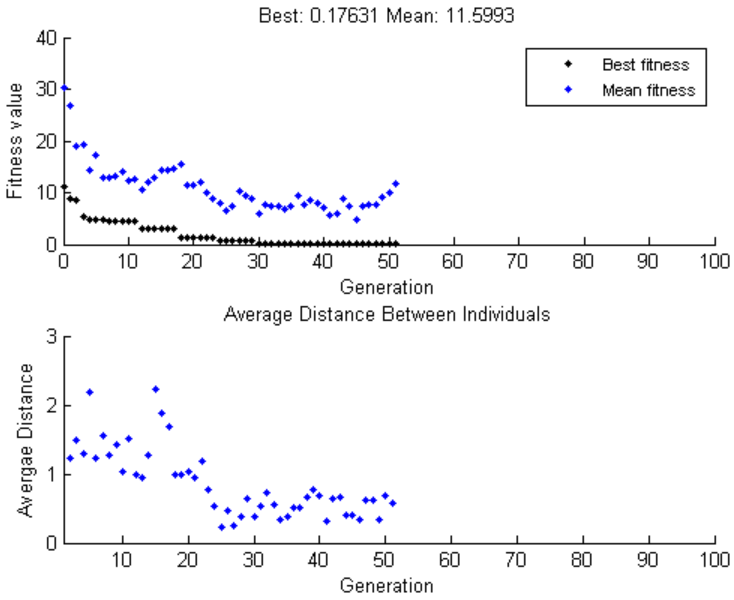
The upper plot, which displays the best fitness at each generation, shows little progress in lowering the fitness value. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too little diversity for the algorithm to make progress.

Next, try setting Initial range to [1;100] and running the algorithm. This time the results are more variable. For example, you might obtain a plot with a best fitness value of 6, as in the following plot.



This time, the genetic algorithm makes progress, but because the average distance between individuals is so large, the best individuals are far from the optimal solution.

Finally, set **Initial range** to [1;2] and run the genetic algorithm. Again, there is variability in the result, but you might obtain a result similar to the following figure. Run the optimization several times, and you eventually obtain a final point near [0;0], with a fitness function value near 0.



The diversity in this case is better suited to the problem, so the genetic algorithm usually returns a better result than in the previous two cases.

Task 3- Setting the Crossover Fraction (Rastrigin's Function)

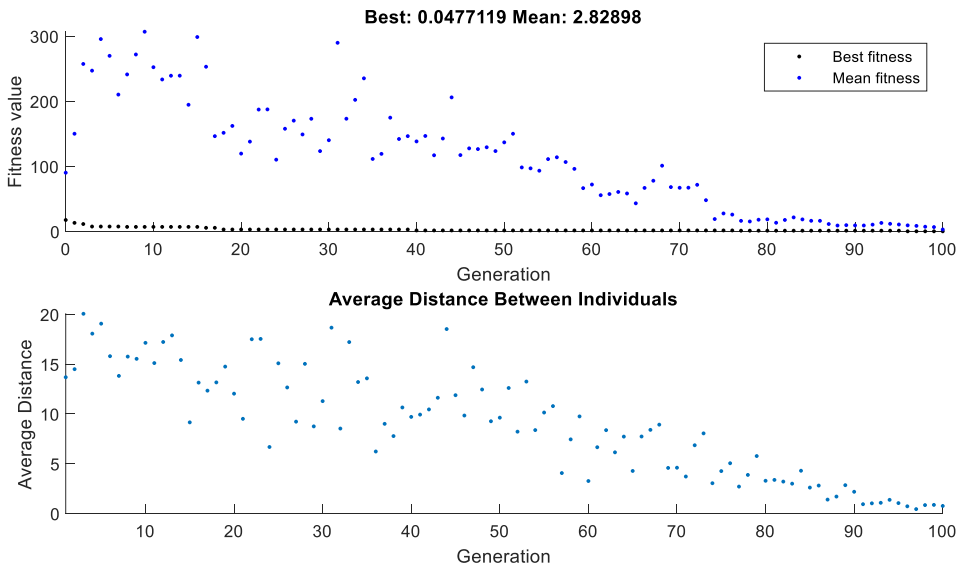
The **Crossover fraction** field, in the **Reproduction** options, specifies the fraction of each population, other than elite children, that are made up of crossover children. A crossover fraction of 1 means that all children other than elite individuals are crossover children, while a crossover fraction of 0 means that all children are mutation children. The following example show that neither of these extremes is an effective strategy for optimizing a function.

To run the example,

- Specify **Stopping criteria / Generations** to 100.
- Select **Best fitness** and **Distance** in the **Plot functions** pane.
- Select the default **Initial range**.

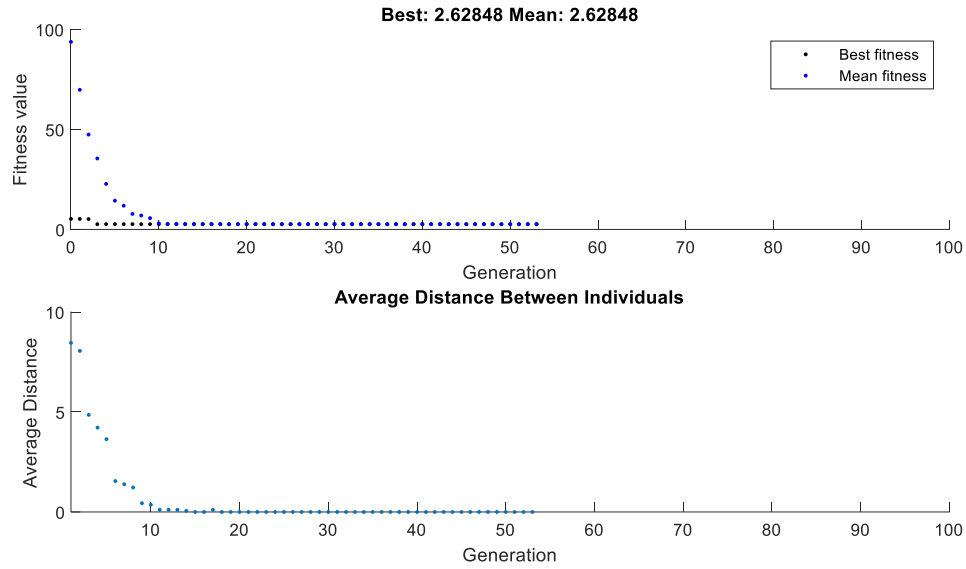
Run the example with the default value of 0.8 for **Crossover fraction**, in the **Options / Reproduction** pane.

Run solver and view results / Start. This returns the best fitness value and displays the following plots.



Crossover without Mutation

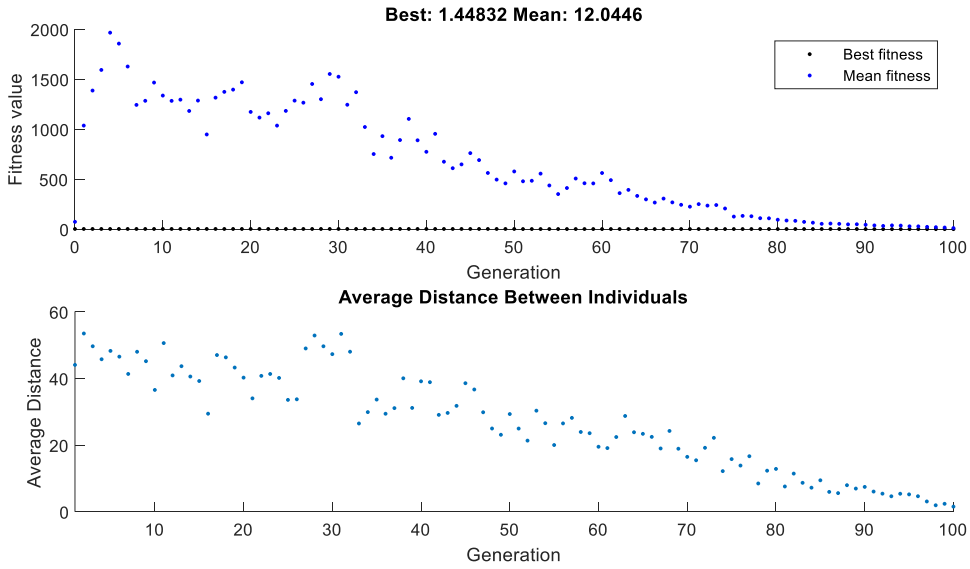
To see how the genetic algorithm performs when there is no mutation, set **Crossover fraction** to 1.0 and click **Start**. This returns the best fitness value in between 2 and 15 and displays the following plots.



In this case, the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation. The algorithm generates the best individual that it can using these genes at generation number 8. After this, it creates new copies of the best individual, which are then selected for the next generation. By generation number 17, all individuals in the population are the same, namely, the best individual. When this occurs, the average distance between individuals is 0. Since the algorithm cannot improve the best fitness value, it stalls after 50 more generations, because **Stall generations** is set to 50.

Mutation without Crossover

To see how the genetic algorithm performs when there is no crossover, set **Crossover fraction** to 0 and click **Start**. This returns the best fitness value and displays the following plots.



In this case, the random changes that the algorithm applies rarely improves the fitness value of the best individual at the first generations. While it improves the individual genes of other individuals, as you can see in the upper plot by

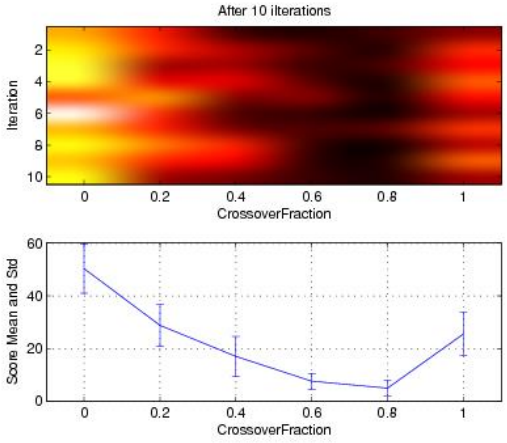
the decrease in the mean value of the fitness function, these improved genes are never combined with the genes of the best individual because there is no crossover. As a result, the best fitness plot usually decreases quite slowly (or is almost a constant line).

Comparing Results for Varying Crossover Fractions

The example deterministicstudy.m, compares the results of applying the genetic algorithm to Rastrigin's function with **Crossover fraction** set to 0, .2, .4, .6, .8, and 1. The example runs for 10 generations. At each generation, the example plots the means and standard deviations of the best fitness values in all the preceding generations, for each value of the **Crossover fraction**.

To run the example, enter deterministicstudy

When the example is finished, the plots appear as in the following figure.

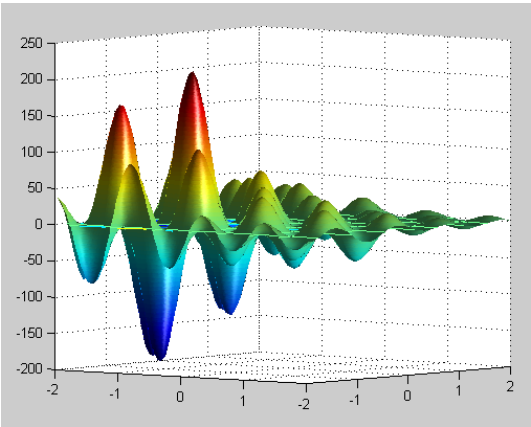


The lower plot shows the means and standard deviations of the best fitness values over 10 generations, for each of the values of the crossover fraction. The upper plot shows a color-coded display of the best fitness values in each generation. For this fitness function, setting **Crossover fraction** to 0.8 yields the best result. However, for another fitness function, a different setting for **Crossover fraction** might yield the best result.

Task 4- SHUFCN – Using the command-line (Shufcn)

SHUFCN is a real valued function of two variables. We can use the function plotobjective in the toolbox to plot the function SHUFCN over the range = [-2 2;-2 2].

Enter: plotobjective(@(x)shufcn(x),[-2 2; -2 2]);



```
FitnessFunction = @(x)shufcn(x);

numberOfVariables = 2;

opts = gaoptimset('PlotFcns',{@gaplotbestf,@gaplotstopping});
opts = gaoptimset(opts,'PopulationSize',10);
```

To specify a different initial range for each variable, the range must be specified as a matrix with two rows and 'numberOfVariables' columns. For example if we set the range to [-1 0; 1 2], then the first variable will be in the range -1 to 1, and the second variable will be in the range 0 to 2 (so each column corresponds to a variable).
The initial range can be specified using gaoptimset. We will pass our options structure 'opts' created above to gaoptimset to modify the value of the parameter 'PopInitRange'.

```
opts = gaoptimset(opts,'PopInitRange',[-2 -2; 2 2]);
```

ga uses four different criteria to determine when to stop the solver. **ga** stops when the maximum number of generations is reached; by default this number is 100*numberOfVariables. **ga** also detects if there is no change in the best fitness value for some time given in seconds (stall time limit), or for some number of generations (stall generations). Another criteria is the maximum time limit in seconds. Here we modify the stopping criteria to increase the maximum number of generations to 150 and the stall generations to 100.

```
opts = gaoptimset(opts,'Generations',150,'StallGenLimit', 100);
```

ga starts with a random set of points in the population and uses operators to produce the next generation of the population. The different operators are scaling, selection, crossover, and mutation. The best function value may improve or it may get worse by choosing different operators. Choosing a good set of operators for your problem is often best done by experimentation.

The toolbox provides several functions to choose from for each operator. Here we choose FITSCALINGPROP for 'FitnessScalingFcn' and SELECTIONTOURNAMENT for 'SelectionFcn'.

```
opts = gaoptimset(opts, 'SelectionFcn',@selectiontournament, 'FitnessScalingFcn',@fitscalingprop);
```

Run the **ga** solver. The first two output arguments returned by **ga** are x, the best point found, and Fval, the function value at the best point. A third output argument, exitFlag tells you the reason why **ga** stopped. **ga** can also return a fourth argument, Output, which contains information about the performance of the solver.

```
[x,Fval,exitFlag,Output] = ga(FitnessFunction,numberOfVariables,[],[],[],[],[],[],[],opts);

fprintf('The number of generations was : %d\n', Output.generations);
fprintf('The number of function evaluations was : %d\n', Output.funccount);
fprintf('The best function value found was : %g\n', Fval);
```

```
formatSpec = 'The best function value was found at point: %7.4f %7.4f \n';
fprintf(formatSpec,x)
```

If you take a look to the struct opts, you can see all the parameters that can be chosen.

Matlab options of all the parameters of GA tool:
http://es.mathworks.com/help/gads/genetic-algorithm-options.html#f9147

Task 5- Binary GA examples

a) maximize the number of 1's in a bit string

In this example we will program a genetic algorithm using MATLAB to maximize the number of 1's in a bit string of length 10. Thus our chromosomes will be binary strings of length 10, and the optimal chromosome that we are searching for is [1111111111].

Since the only possibilities for each bit of a chromosome are 1 or 0, the number of 1's in a chromosome is equivalent to the sum of all the bits. Since Matlab genetic algorithm minimize the fitness function, we set it to: -sum(x).

Set the **Fitness function**:

```
FitnessFunction = @(x)-sum(x);
numberOfVariables = 10;
```

Set the **Plot funcions**:

```
opts = gaoptimset('PlotFcns',{@gaplotbestf,@gaplotstopping});
```

Set the **Population size** to 10, the **Number of generations** to 50, the **Stall generations limit** to 50, the **Selection function** to tournament and the **Fitness scaling function** to proportional.

Note: Remember that you can take a look to the different options available for each parameter and their meaning in the right hand side of the optimization app panel, pushing the >> symbol.

```
opts = gaoptimset(opts,'PopulationSize',20);
opts = gaoptimset(opts,'Generations',50,'StallGenLimit', 50);
opts = gaoptimset(opts, 'SelectionFcn',@selectiontournament, 'FitnessScalingFcn',@fitscalingprop);
```

Set the **Population Type** to 'bitstring', since in this example the type of the input is binary. Set the **Output functions** to @my_view (included in the .zip file). Output functions are functions that the genetic algorithm calls at each iteration. In this case my_view is a function that displays in the Matlab command line the population in each iteration.

```
opts = gaoptimset(opts,'PopulationType', 'bitstring', 'OutputFcns',@my_view);
```

Run the GA:

```
[x,Fval,exitFlag,Output,population,scores] = ga(FitnessFunction,numberOfVariables,[],[],[],[],[],[],[],opts);
```

Print some information:

```
fprintf('The number of generations was : %d\n', Output.generations);
fprintf('The number of function evaluations was : %d\n', Output.funccount);
fprintf('The best function value found was : %g\n', Fval);
```

You can see the evolution of the population in the command window. As you can see the genetic algorithm is able to arrive to the optimal solution very quickly. You can play with the different parameters: population size (e.g. 10, 20,

50) and number of generations (g.e. 20, 50, 150) to determine the simplest genetic algorithm that gives good results to this problem.

b) find a bit string containing exactly *k* 1's

This example is a generalization of the previous one in which we are seeking bit strings with exactly *k* 1's in a space formed by all bit strings of length 20. Thus our chromosomes will be binary strings of length 20 and the number of optimal solutions will depend on the particular choice of *k* (many for the central values of k and less as the extremes 0 and 20 are approached). Try, for instance, *k*=1, 2 and 3 and compare the number of generations you need to reach an optimal fitness solution in each case.

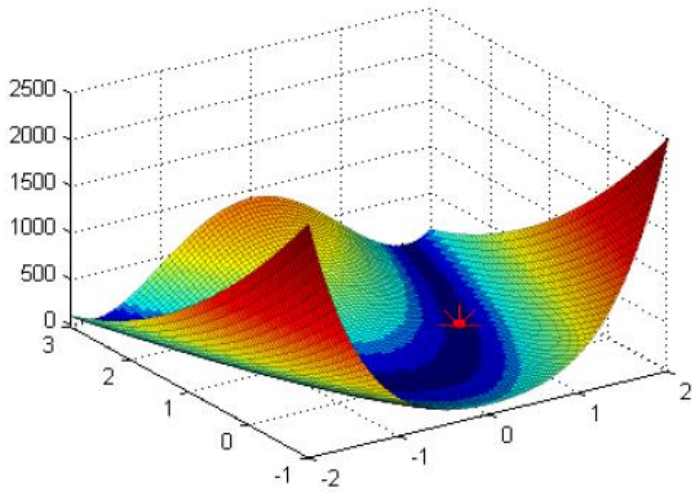
For this example, set the **Fitness function**:

```
FitnessFunction = @(x)abs(sum(x)-k);  % replace k by the selected constant between 0 and 20
numberOfVariables = 20;
```

Task 6- Rosenbrock Exercise: TO BE DELIVERED

In mathematical optimization, the Rosenbrock function is a non-convex function used as a performance test problem for optimization algorithms introduced by Howard H. Rosenbrock in 1960. The global minimum is inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial. To converge to the global minimum, however, is difficult.

f(x, y) = (1-x)^2 + 100(y-x^2)^2



Exercise: Find the minimum of the Rosenbrock function by comparing different values of the following operators: population size, number of generations, initial range, selection and reproduction (crossover and mutation). Write a small document with your names on it, enumerating the different sets of parameters tested and your own conclusions for this experiment (at most three pages). You can include plots if you like, but it is not mandatory. Answer the questions: Where is the global minimum? Which is the global minimum? Which is the parameters combination that give you the best result? You should upload a zip file in the Raco including this document together with the Matlab script .m file you used to obtain the results. You have time until Thursday 23rd of December included.

```
FitnessFunction = @(x)(1-x(1))^2+100*(x(2)-x(1)^2)^2;
```

Note: The command-line interface enables you to run the genetic algorithm many times, with different options settings, using a file. For example, you can run the genetic algorithm with different settings for Crossover fraction to see which one gives the best results. The following code runs the function **ga** 21 times, varying CrossoverFraction from 0 to 1 in increments of 0.05, and records the results.

```
opts = gaoptimset('Generations',300,'Display','none');
opts = gaoptimset(opts, 'PopInitRange',[-2 -2; 2 2]);
rng default % rng (random number generation) for reproducibility it takes the same random number
record=[];
for n=0:.05:1
   opts = gaoptimset(opts,'CrossoverFraction',n);
   [x fval]=ga(FitnessFunction,2,[],[],[],[],[],[],[],opts);
   record = [record; fval];
end
```

You can plot the values of fval against the crossover fraction with the following commands:

```
plot(0:.05:1, record);
xlabel('Crossover Fraction');
ylabel('fval')
```

You can get a smoother plot of fval as a function of the crossover fraction by running **ga** 20 times and averaging the values of fval for each crossover fraction.