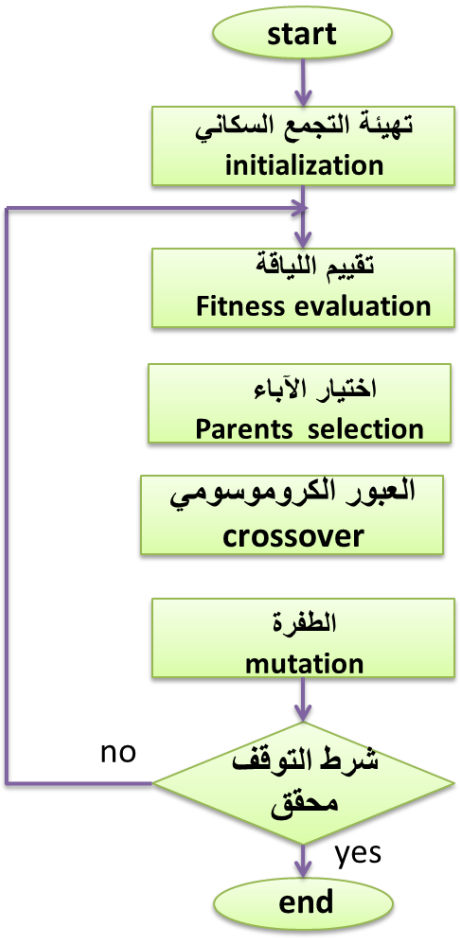
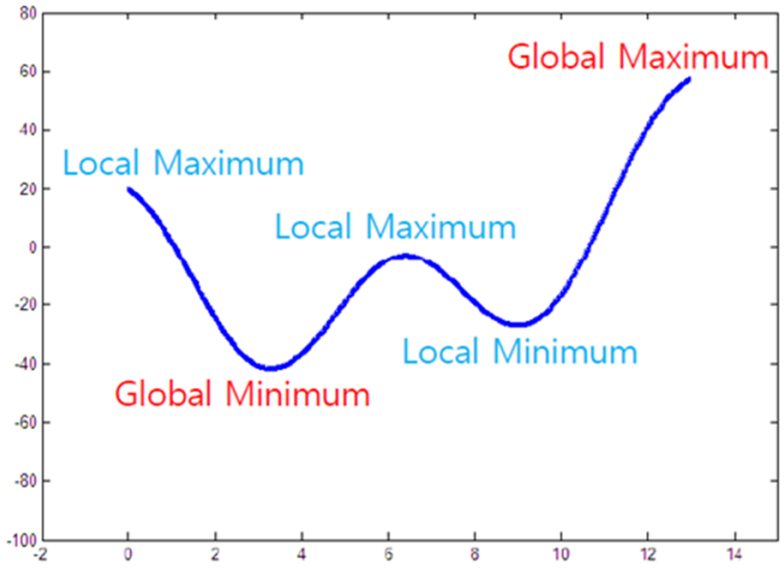
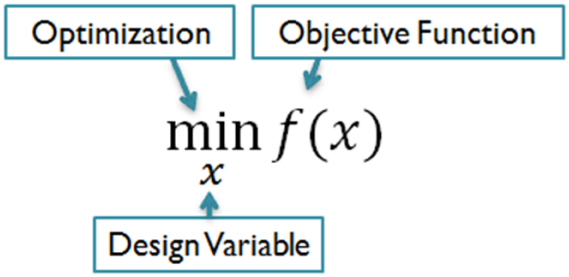
مسائل إيجاد الحل الأمثل باستخدام الخوارزميات الجينية

إيجاد الحل الأمثل(Optimization) يتعامل مع مسائل إيجاد القيمة الصغرى أو القيمة الكبرى لتابع بعدة متحولات و غالباً ما تكون هذه المسائل خاضعة لضوابط (قيود Constraints) على شكل مساواة و / أو عدم مساواة (متراجحات).







حل مسائل الـ optimization بوجود قيود (Constraints) في الماتلاب

$$\min_x f(x)$$

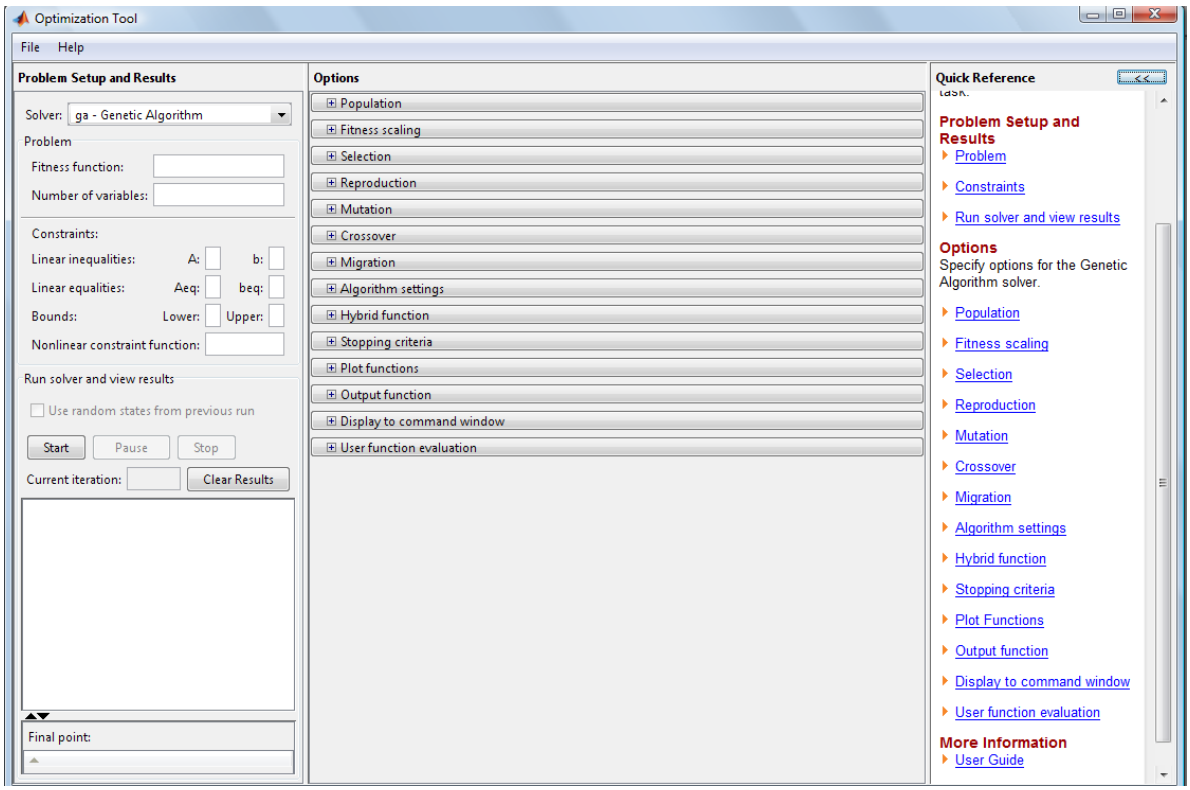
Linear Constraints

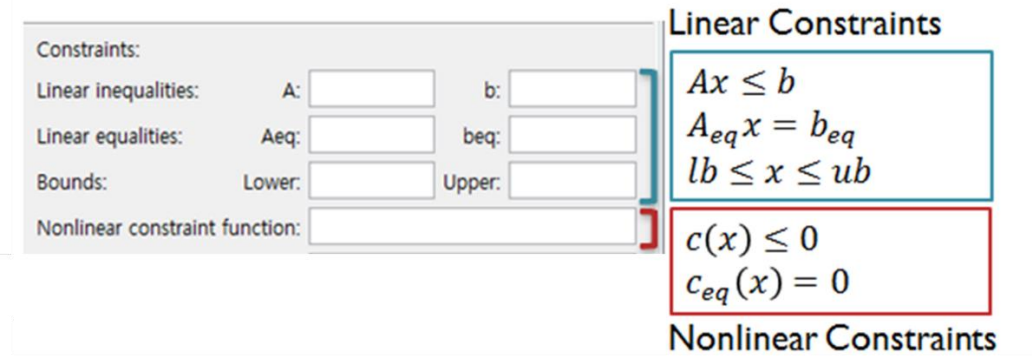
$$Ax \leq b$$
$$A_{eq}x = b_{eq}$$
$$lb \leq x \leq ub$$

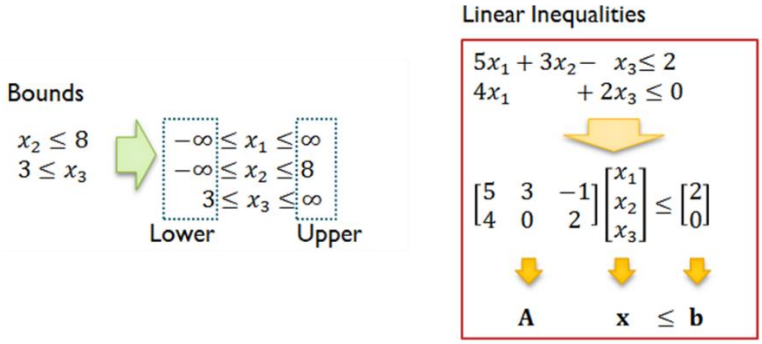
$$c(x) \leq 0$$
$$c_{eq}(x) = 0$$

Nonlinear Constraints

Optimtool


Optimization Tool
File Help
Problem Setup and Results
Solver: ga - Genetic Algorithm
Problem
Fitness function:
Number of variables:
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function:
Run solver and view results
Use random states from previous run
Start Pause Stop
Current iteration: Clear Results
Final point:
Options
Population
Fitness scaling
Selection
Reproduction
Mutation
Crossover
Migration
Algorithm settings
Hybrid function
Stopping criteria
Plot functions
Output function
Display to command window
User function evaluation
Quick Reference
Problem Setup and Results
Problem
Constraints
Run solver and view results
Options
Specify options for the Genetic Algorithm solver.
Population
Fitness scaling
Selection
Reproduction
Mutation
Crossover
Migration
Algorithm settings
Hybrid function
Stopping criteria
Plot Functions
Output function
Display to command window
User function evaluation
More Information
User Guide


Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function:
Linear Constraints
$Ax \leq b$
$A_{eq}x = b_{eq}$
$lb \leq x \leq ub$
$c(x) \leq 0$
$c_{eq}(x) = 0$
Nonlinear Constraints


Bounds
$x_2 \leq 8$
$3 \leq x_3$
$-\infty \leq x_1 \leq \infty$
$-\infty \leq x_2 \leq 8$
$3 \leq x_3 \leq \infty$
Lower
Upper
Linear Inequalities
$5x_1 + 3x_2 - x_3 \leq 2$
$4x_1 + 2x_3 \leq 0$
$\begin{bmatrix} 5 & 3 & -1 \\ 4 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 0 \end{bmatrix}$
A
x
≤ b

Example

We want to minimize a simple fitness function of two variables x1 and x2

```
min f(x) = 100 * (x1^2 - x2) ^2 + (1 - x1)^2;
 x
```

such that the following two nonlinear constraints and bounds are satisfied

```
x1*x2 + x1 - x2 + 1.5 <=0,  (nonlinear constraint)
10 - x1*x2 <=0,             (nonlinear constraint)
0 <= x1 <= 1, and           (bound)
0 <= x2 <= 13               (bound)
```

```
function [c, ceq] = simple_constraint(x)
c = [1.5 + x(1)*x(2) + x(1) - x(2);
-x(1)*x(2) + 10];
ceq = [];
```

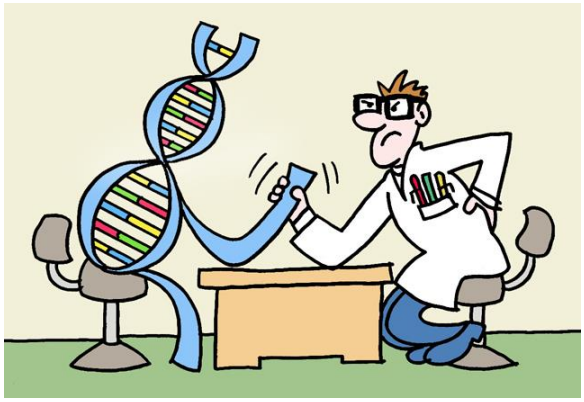
```
ObjectiveFunction = @simple_fitness;
nvars = 2;     % Number of variables
LB = [0 0];    % Lower bound
UB = [1 13];   % Upper bound
ConstraintFunction = @simple_constraint;
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB, ...
    ConstraintFunction)
```

```
x =

    0.8122   12.3122


fval =

  1.3578e+004
```

بما أن صندوق أدوات الخوارزميات الجينية المستخدم في الماتلاب يعمل فقط على إيجاد القيمة الصغرى لتابع، فكيف يمكن استخدامه عندما تكون المسألة هي إيجاد القيمة الكبرى للتابع؟

$$\max_x f(x) = \min_x [-f(x)]$$

تحسينات إضافية على الخوارزميات الوراثية



معايرة تابع التقييم

- تعتبر دالة التقييم من أهم عمليات الخوارزميات الوراثية. وسوء اختيار هذه الدالة يؤثر سلباً على أداء عملية الاستقصاء
- إن لياقة الكروموسوم مقارنة بالمعدل العام للياقة الجيل هي التي تحدد فرصة الانتقاء وبالتالي إذا كانت لياقة كروموسوم ما ثلاثة أضعاف معدل اللياقة فإن هذا الكروموسوم قد يفرز ثلاثة نسخ في الجيل التالي
- أما إذا كانت جميع اللياقات متقاربة (نتيجة سوء اختيار تابع التقييم) فستصبح عملية الانتقاء بدون فعالية كمثال على ذلك، يحتوي الجدول على قيم لياقة خمسة كروموسومات كلها قريبة جداً من معدل اللياقة وبالتالي كلها متقاربة مما يحد من فعالية الانتقاء

الكروموسوم	1	2	3	4	5
اللياقة	100.320	100.007	100.991	100.215	100.075

لحل هذه المشكلة، يمكن اللجوء إلى معايرة تابع التقييم(Normalization) فلو خصمنا 100 من قيمة اللياقة المدرجة في الجدول السابق نلاحظ أن عملية الانتقاء تتم في ظروف أفضل بكثير وتعكس لياقة الكروموسومات بشكل أصح

الكروموسوم	1	2	3	4	5	المعدل
اللياقة الأصلية	100.320	100.007	100.991	100.215	100.075	100.3216
معايرة اللياقة بخصم 100	0.320	0.007	0.991	0.215	0.075	0.3216

عوضاً عن خصم 100 من قيمة اللياقة في المثال السابق كان من الممكن ترتيب الكروموسومات من الأحسن إلى الأسوأ ثم إعطائها لياقة جديدة كأن نعطي 10 لأحسنها ثم 8 للثاني و 6 للثالث و 4 للرابع وأخيراً 2 لأسوئها. أو أي طريقة معايرة أخرى نراها مناسبة.

المهم في كل هذا هو أن نراعي عاملين أساسيين:

- الأول هو أن لا تكون كل اللياقات متقاربة من المعدل العام
- والثاني هو أن لا يطغى كروموسوم وحيد على كل الكروموسومات الأخرى و يحرمها تماماً من الاستمرار

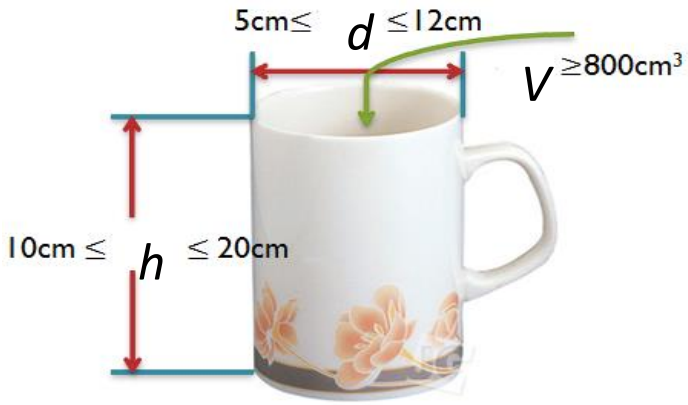
النخبوية (Elitism)

- عند تطبيق الخوارزميات الوراثية بشكلها الكلاسيكي قد يكون من الوارد أن تعجز بعض الكروموسومات الجيدة عن الاستمرار نظراً لعشوائية الكثير من العمليات.
- رغم ندرة هذه الحالة، بإمكاننا ضمان استمرارية الكروموسومات الجيدة باستعمال طريقة النخبوية
- في هذه الطريقة، يتم نقل الكروموسومات الجيدة إلى الجيل التالي دون أن نطبق عليها أي من عمليات الخوارزميات الوراثية في حين تمرّ الكروموسومات الأخرى بكل العمليات
- من إيجابيات هذه الطريقة هي الزيادة في فعالية وسرعة الخوارزميات، لكنها بالمقابل تزيد من إمكانية طغيان كروموسوم واحد على بقية الكروموسومات

مسألة إيجاد الحل الأمثل مع وجود قيود

Optimization Problem with Constraint

نريد تقليل تكلفة صناعة كأس من الزجاج مع احترام القيود التالية:



Constraints

$$5 \le d \le 12$$
$$10 \le h \le 20$$
$$V(d,h) = \frac{\pi}{4} d^2 h \ge 800$$

وبالتالي تقليل التكلفة يعني تصغير:

مساحة الكأس:

Objective Function

$$S(d,h) = \frac{\pi}{4} d^2 + \pi d h$$

Genetic Algorithm and Direct Search Toolbox

The Genetic Algorithm and Direct Search Toolbox is a collection of functions that extend the capabilities of the Optimization Toolbox and theMATLAB numeric computing environment. The Genetic Algorithm and Direct Search Toolbox includes routines for solving optimization problems using

• Genetic algorithm

• Direct search

These algorithms enable you to solve a variety of optimization problems that lie outside the scope of the Optimization Toolbox.

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

• *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation.

• *Crossover rules* combine two parents to form children for the next generation.

• *Mutation rules* apply random changes to individual parents to form children

The genetic algorithm at the command line, call the genetic algorithm function ga with the syntax

```
[x fval] = ga(@fitnessfun, nvars, options)
```

where

- @fitnessfun is a handle to the fitness function.
- nvars is the number of independent variables for the fitness function.
- options is a structure containing options for the genetic algorithm. If you do not

pass in this argument, 'ga' uses its default options.

The results are given by

- x— Point at which the final value is attained
- fval—Final value of the fitness function

ga

Find minimum of function using genetic algorithm

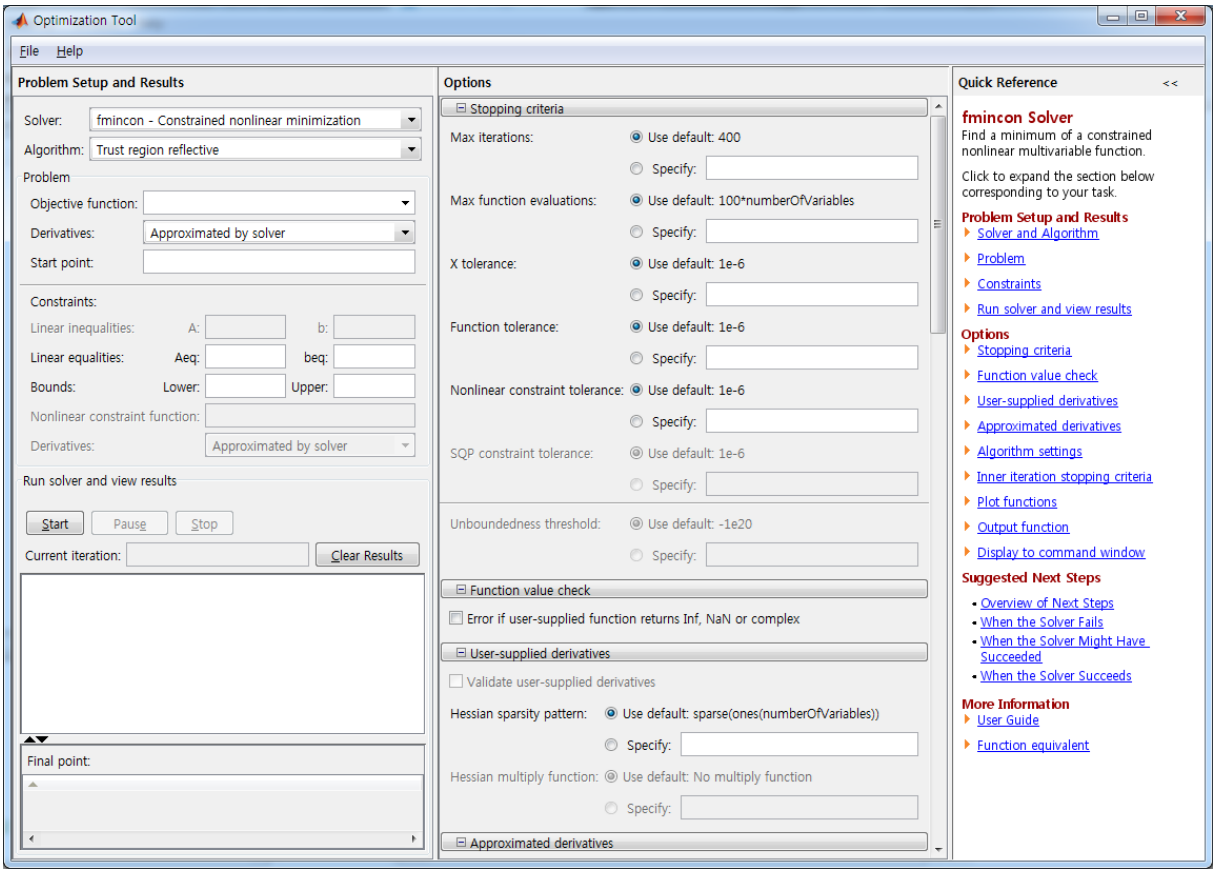
Syntax

```
x = ga(fitnessfcn,nvars)
x = ga(fitnessfcn,nvars,A,b)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon)
x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)
x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon)
x = ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)
x = ga(problem)
[x,fval] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output,population] = ga(fitnessfcn,nvars,...)
[x,fval,exitflag,output,population,scores] = ga(fitnessfcn,nvars,...)
```

Genetic Algorithm Optimizations Using the Optimization Tool GUI

To open the Optimization Tool, enter

`optimtool('ga')` at the command line, or enter `optimtool` and then choose `ga` from the **Solver** menu.



To use the Optimization Tool, you must first enter the following information:

- **Fitness function** — The objective function you want to minimize. Enter the fitness function in the form `@fitnessfun`, where fitnessfun.m is an M-file that computes the fitness function. The @ sign creates a function handle to `fitnessfun`.
- **Number of variables** — The length of the input vector to the fitness function. you would enter 2.

You can enter constraints or a nonlinear constraint function for the problem in the **Constraints** pane. If the problem is unconstrained, leave these fields blank.

To run the genetic algorithm, click the **Start** button. The tool displays the results of the optimization in the **Run solver and view results** pane.

You can change the options for the genetic algorithm in the **Options** pane. To view the options in one of the categories listed in the pane, click the + sign next to it.

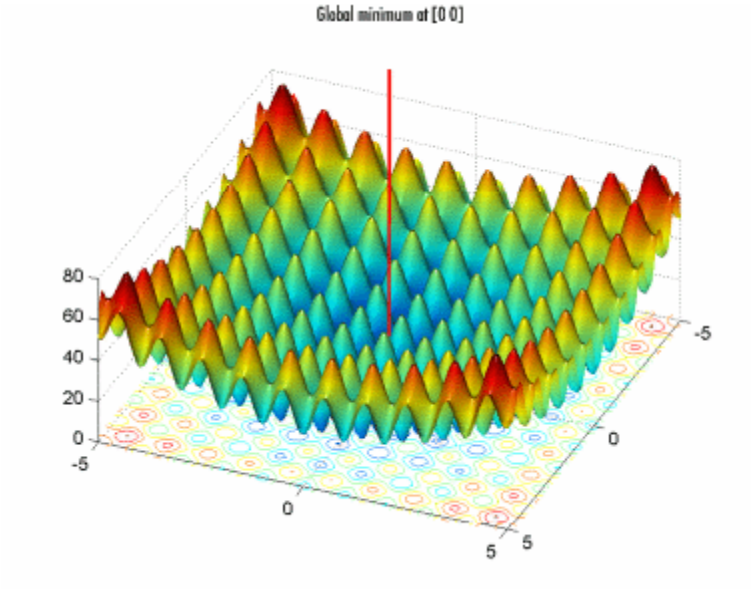
Example 1 — Finding the Minimum of the Rastrigin's Function

This example shows how to 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

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

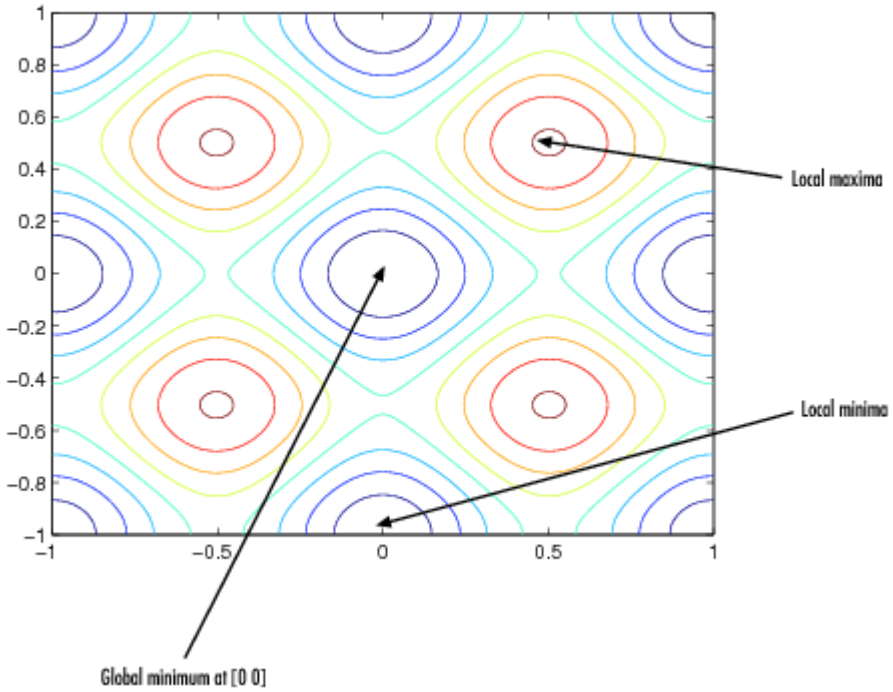
The toolbox contains an M-file, rastriginsfcn.m, that computes the values of Rastrigin's function. The following figure shows a plot of Rastrigin's function.



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.

Rastrigin's function is often used to test the genetic algorithm, because its many local minima make it difficult for standard, gradient-based methods to find the global minimum.

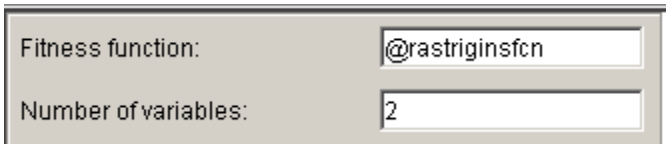
The following contour plot of Rastrigin's function shows the alternating maxima and minima.



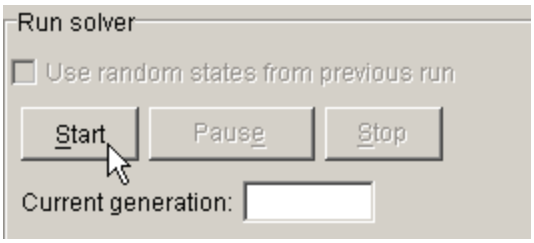
To find the minimum, do the following steps:

1. Enter gatool at the command line to open the Genetic Algorithm Tool.
2. Enter the following in the Genetic Algorithm Tool:
   - In the **Fitness function** field, enter `@rastriginsfcn`.
   - In the **Number of variables** field, enter 2, the number of independent variables for Rastrigin's function.

     The **Fitness function** and **Number of variables** fields should appear as shown in the following figure.



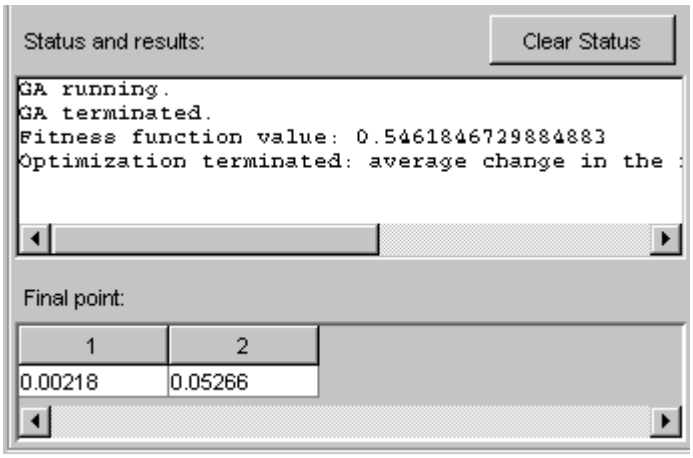
3. Click the **Start** button in the **Run solver** pane, as shown in the following figure.



While the algorithm is running, the **Current generation** field displays the number of the current generation. You can temporarily pause the algorithm by clicking the **Pause** button. When you do so, the button name changes to **Resume**. To resume the algorithm from the point at which you paused it, click **Resume**.

When the algorithm is finished, the **Status and results** pane appears as shown in the following figure.

The **Status and results** pane displays the following information:



- The final value of the fitness function when the algorithm terminated:
- `Function value: 0.5461846729884883`

  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.TolFun.`
- The final point, which in this example is `[0.00218 0.05266]`.

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

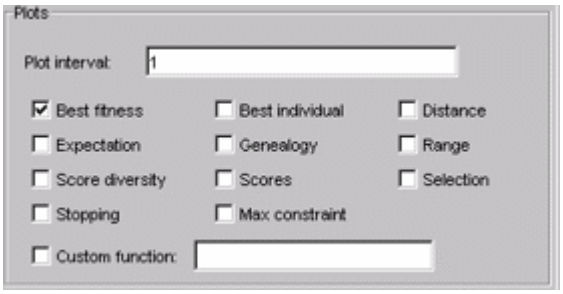
```
Optimization terminated:
average change in the fitness value less than options.TolFun.

x =

    0.0229    0.0106


fval =

    0.1258


exitflag =

1
```

where

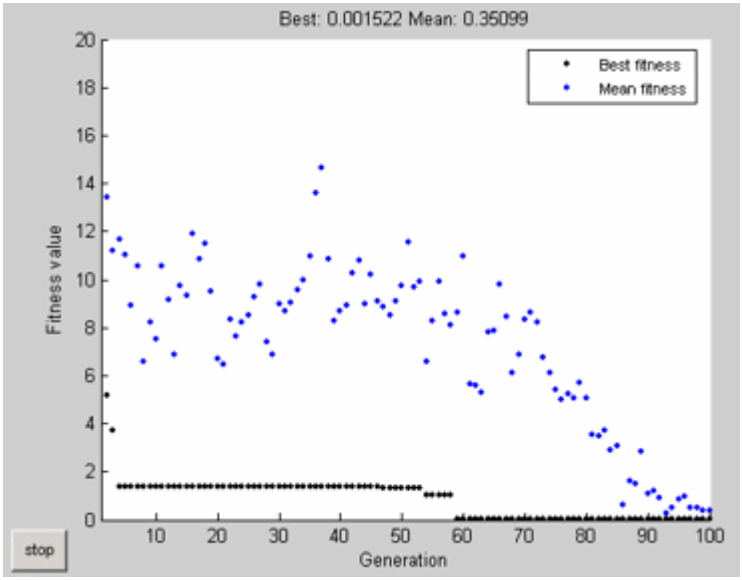
- x is the final point returned by the algorithm.
- `fval` is the fitness function value at the final point.
- `exitflag` is integer value corresponding to the reason that the algorithm terminated.

Displaying Plots

The **Plots** pane 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 the box next to **Best fitness**, as shown in the following figure.



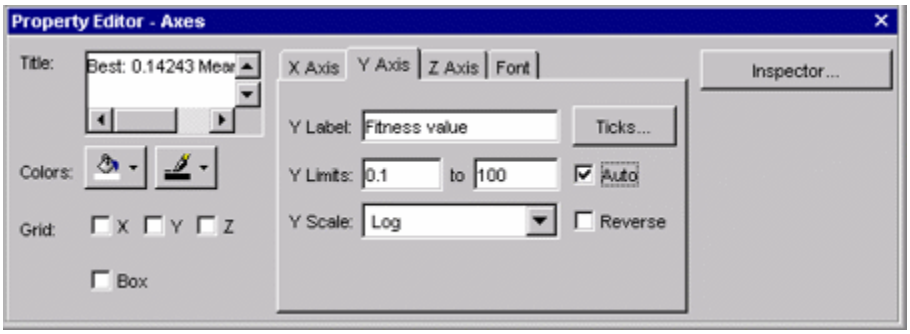
When you click **Start**, the Genetic Algorithm Tool displays a plot of the best and mean values of the fitness function at each generation. When the algorithm stops, the plot appears as shown in the following figure.



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.

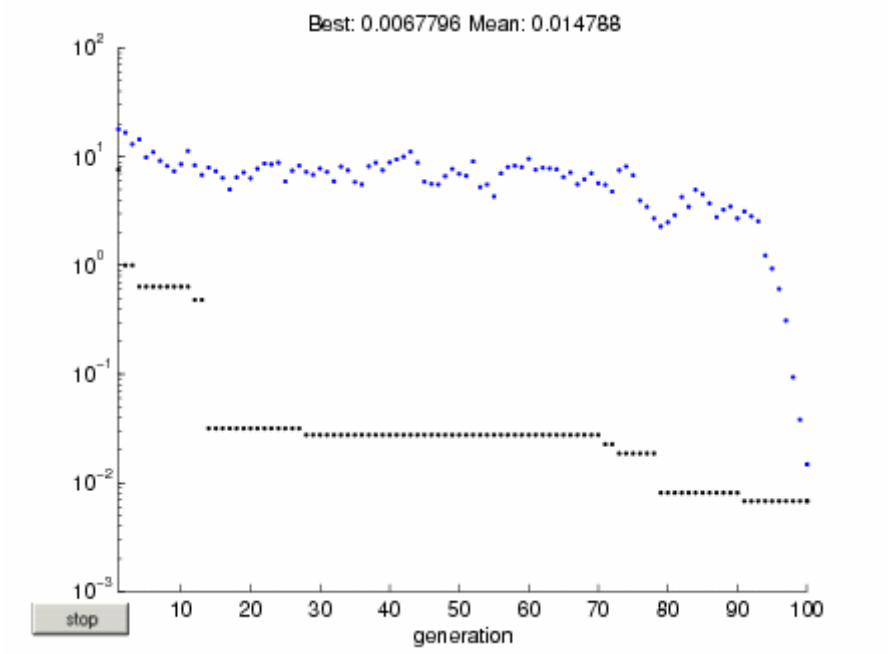
To get a better picture of how much the best fitness values are decreasing, you can change the scaling of the *y*-axis in the plot to logarithmic scaling. To do so,

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



2. Click the **Y** tab.
3. In the **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.

Example2–– Minimizing a Function Using the Genetic Algorithm

Here we want to minimize a simple function of two variables

```
min f(x) = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;
x
```

Coding the Fitness Function

We create an M-file named simple_fitness.m with the following code in it:

```
function y = simple_fitness(x)
y = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;
```

The Genetic Algorithm solver assumes the fitness function will take one input x where x is a row vector with as many elements as number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument y.

Minimizing Using GA

To minimize our fitness function using the GA function, we need to pass in a function handle to the fitness function as well as specifying the number of variables in the problem.

```
FitnessFunction = @simple_fitness;
numberOfVariables = 2;
[x,fval] = ga(FitnessFunction,numberOfVariables)
Optimization terminated: average change in the fitness value
less than options.TolFun.

x =

    0.9652    0.9340


fval =

    0.0017
```

The x returned by the solver is the best point in the final population computed by GA. The fval is the value of the function @simple_fitness evaluated at the point x.

Constrained Minimization Using the Genetic Algorithm

We want to minimize a simple fitness function of two variables x1 and x2

```
min f(x) = 100 * (x1^2 - x2) ^2 + (1 - x1)^2;
  x
```

such that the following two nonlinear constraints and bounds are satisfied

```
x1*x2 + x1 - x2 + 1.5 <=0,  (nonlinear constraint)
10 - x1*x2 <=0,             (nonlinear constraint)
0 <= x1 <= 1, and           (bound)
0 <= x2 <= 13               (bound)
```

Coding the Fitness Function

We create an M-file named simple_fitness.m with the following code in it:

```
function y = simple_fitness(x)
  y = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;
```

The Genetic Algorithm (GA) function assumes the fitness function will take one input x where x has as many elements as number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument y.

Coding the Constraint Function

We create an M-file named simple_constraint.m with the following code in it:

```
function [c, ceq] = simple_constraint(x)
c = [1.5 + x(1)*x(2) + x(1) - x(2);
-x(1)*x(2) + 10];
ceq = [];
```

The GA function assumes the constraint function will take one input x where x has as many elements as number of variables in the problem. The constraint function computes the values of all the inequality and equality constraints and returns two vectors c and ceq respectively.

Minimizing Using GA

To minimize our fitness function using the GA function, we need to pass in a function handle to the fitness function as well as specifying the number of variables as the second argument. Lower and upper bounds are provided as LB and UB respectively. In addition, we also need to pass in a function handle to the nonlinear constraint function.

```
ObjectiveFunction = @simple_fitness;
nvars = 2;     % Number of variables
LB = [0 0];    % Lower bound
UB = [1 13];   % Upper bound
ConstraintFunction = @simple_constraint;
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB, ...
    ConstraintFunction)
Optimization terminated: average change in the fitness value
less than options.TolFun
 and constraint violation is less than options.TolCon.

x =

    0.8122   12.3122


fval =

  1.3578e+004
```

Note that for our constrained minimization problem, the GA function changed the mutation function to @mutationadaptfeasible. The default mutation function, @mutationgaussian, is only appropriate for unconstrained minimization problems.

GA Operators for Constrained Minimization

The GA solver handles linear constraints and bounds differently from nonlinear constraints. All the linear constraints and bounds are satisfied throughout the optimization. However, GA may not satisfy all the nonlinear constraints at every generation. If GA converges to a solution, the nonlinear constraints will be satisfied at that solution.

GA uses the mutation and crossover functions to produce new individuals at every generation. The way the GA satisfies the linear and bound constraints is to use mutation and crossover functions that only generate feasible points. For example, in the previous call to GA, the default mutation function MUTATIONGAUSSIAN will not satisfy the linear constraints and so the MUTATIONADAPTFEASIBLE is used instead. If you provide a custom mutation function, this custom function must only generate points that are feasible with respect to the linear and bound constraints. All the crossover functions in the toolbox generate points that satisfy the linear constraints and bounds.

We specify MUTATIONADAPTFEASIBLE as the mutation function for our minimization problem by using GAOPTIMSET function.

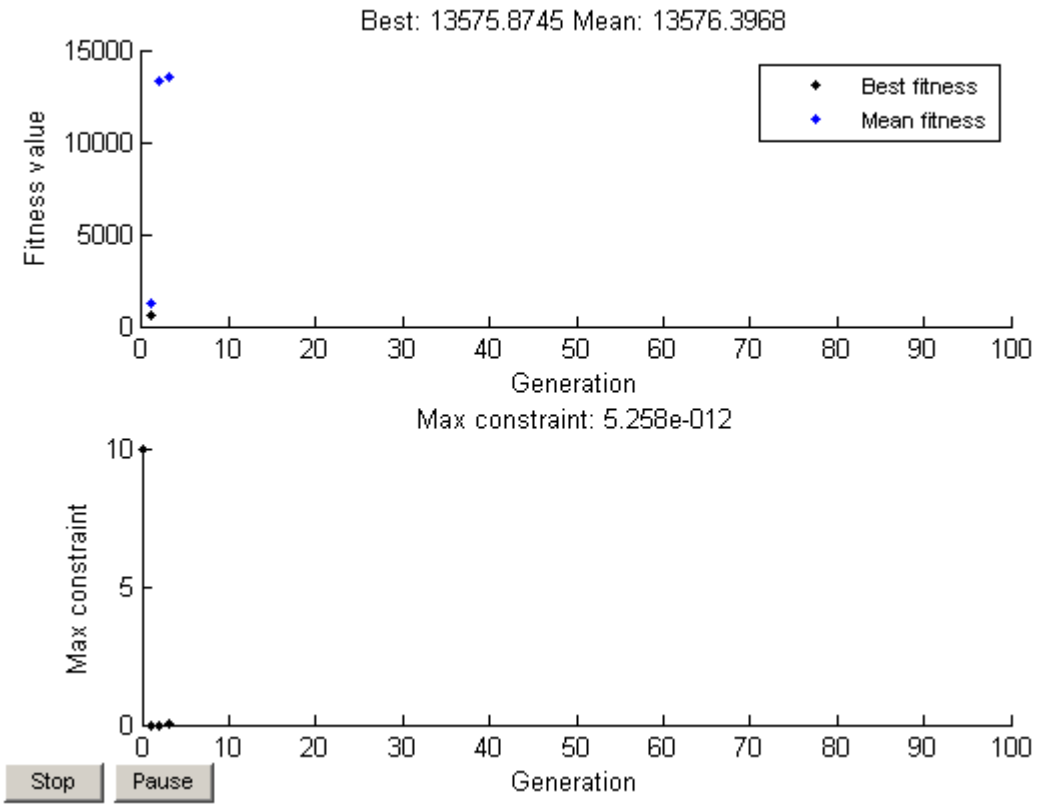
```
options = gaoptimset('MutationFcn',@mutationadaptfeasible);
% Next we run the GA solver.
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB, ...
    ConstraintFunction,options)
Optimization terminated: average change in the fitness value
less than options.TolFun
 and constraint violation is less than options.TolCon.

x =

    0.8122   12.3122


fval =

  1.3578e+004
```

Adding Visualization

Next we use GAOPTIMSET to create an options structure to select two plot functions. The first plot function is GAPLOTBESTF, which plots the best and mean score of the population at every generation. The second plot function is GAPLOTMAXCONSTR, which plots the maximum constraint violation of nonlinear constraints at every generation. We can also visualize the progress of the algorithm by displaying information to the command window using the 'Display' option.

```
options =
gaoptimset(options,'PlotFcns',{@gaplotbestf,@gaplotmaxconstr},
...
    'Display','iter');
```

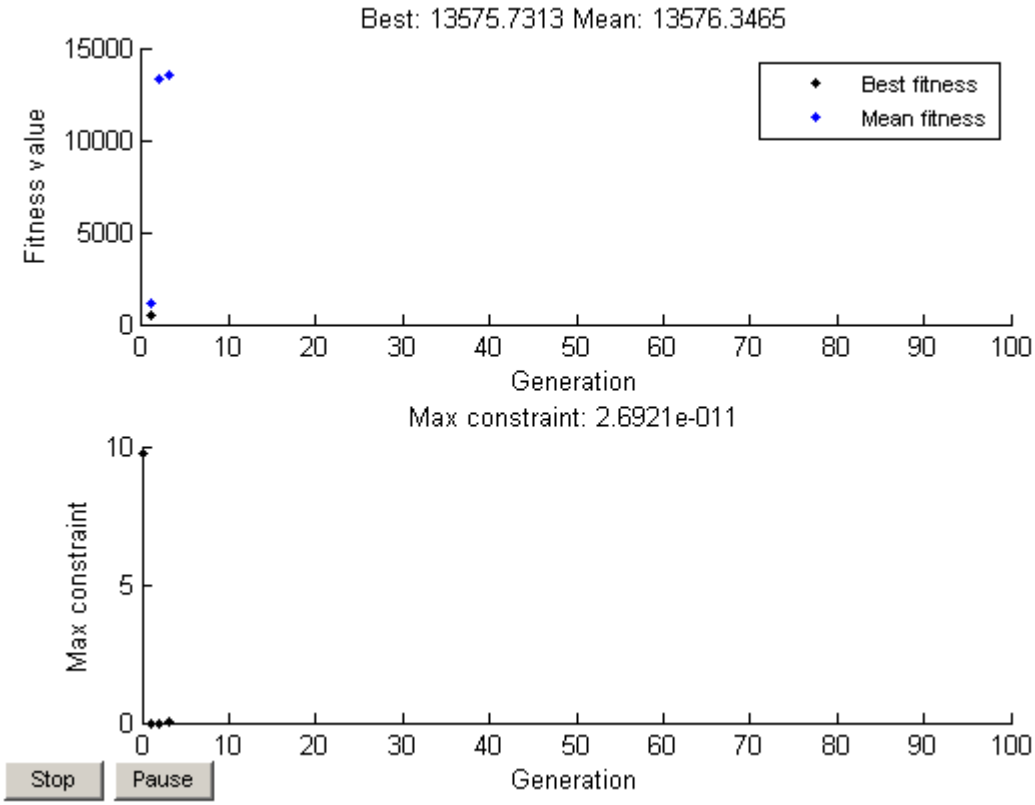
```
% Next we run the GA solver.
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB, ...
    ConstraintFunction,options)
                                Best       max        Stall
Generation  f-count          f(x)     constraint  Generations
    1         1080        13596.7             0       0
    2         2136        13578.2             0       0
    3         3188        13578.2   5.258e-012        0
Optimization terminated: average change in the fitness value
less than options.TolFun
 and constraint violation is less than options.TolCon.

x =

    0.8122   12.3122


fval =

  1.3578e+004
```



Providing a Start Point

A start point for the minimization can be provided to GA function by specifying the InitialPopulation option. The GA function will use the first individual from InitialPopulation as a start point for a constrained minimization.

```
X0 = [0.5 0.5]; % Start point (row vector)
options = gaoptimset(options,'InitialPopulation',X0);
% Next we run the GA solver.
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB, ...
    ConstraintFunction,options)
                            Best       max        Stall
Generation  f-count         f(x)     constraint  Generations
    1         1084       13578.4             0      0
    2         2140       13578.2             0      0
    3         3192       13578.2   2.692e-011       0
Optimization terminated: average change in the fitness value
less than options.TolFun
 and constraint violation is less than options.TolCon.

x =

    0.8122   12.3122


fval =

  1.3578e+004
```



Genetic Algorithm Options

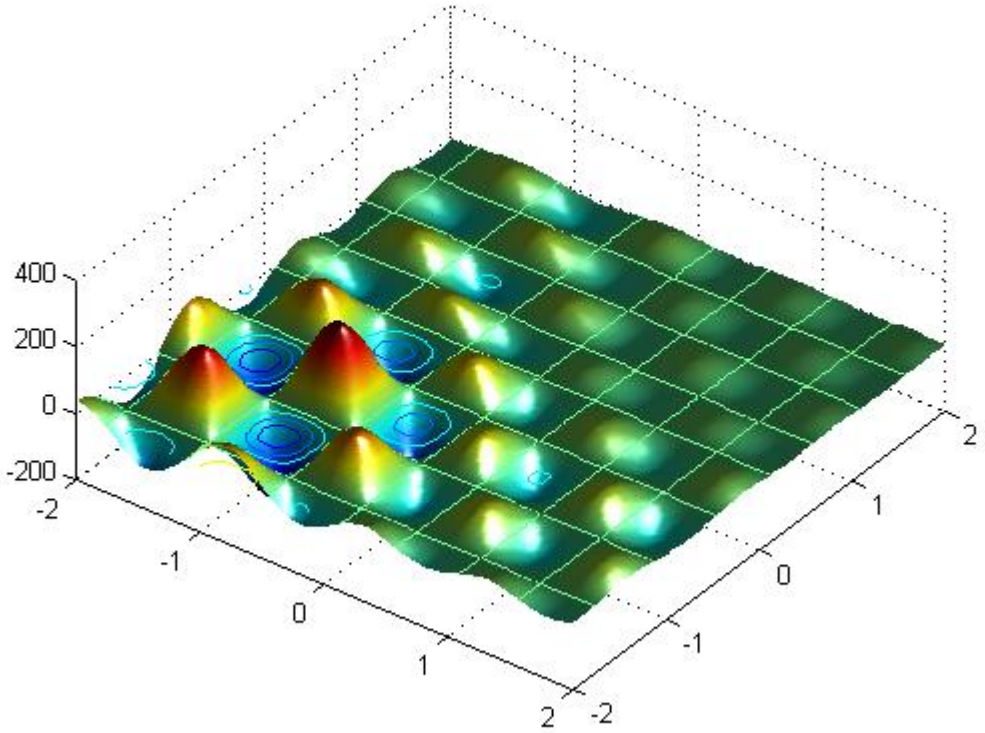
This is a demonstration of how to create and manage options for the genetic algorithm function GA using GAOPTIMSET in the Genetic Algorithm and Direct Search Toolbox.

Setting Up a Problem for GA

GA searches for a minimum of a function using the genetic algorithm. For this demo we will use GA to minimize the fitness function 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].

```
plotobjective(@shufcn,[-2 2; -2 2]);
```



To use the GA solver, we need to provide at least two input arguments, a fitness function and the number of variables in the problem. 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.

```
FitnessFunction = @shufcn;
numberOfVariables = 2;
```

Run the GA solver.

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

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

```
The number of generations was : 51
The number of function evaluations was : 1040
The best function value found was : -185.379
```

Note that when you run this demo, your result may be different from the results shown; This will be explained in a section later in this demo.

How the Genetic Algorithm Works

The Genetic Algorithm (GA) works on a population using a set of operators that are applied to the population. A population is a set of points in the design space. The initial population is generated randomly by default. The next generation of the population is computed using the fitness of the individuals in the current generation.

Adding Visualization

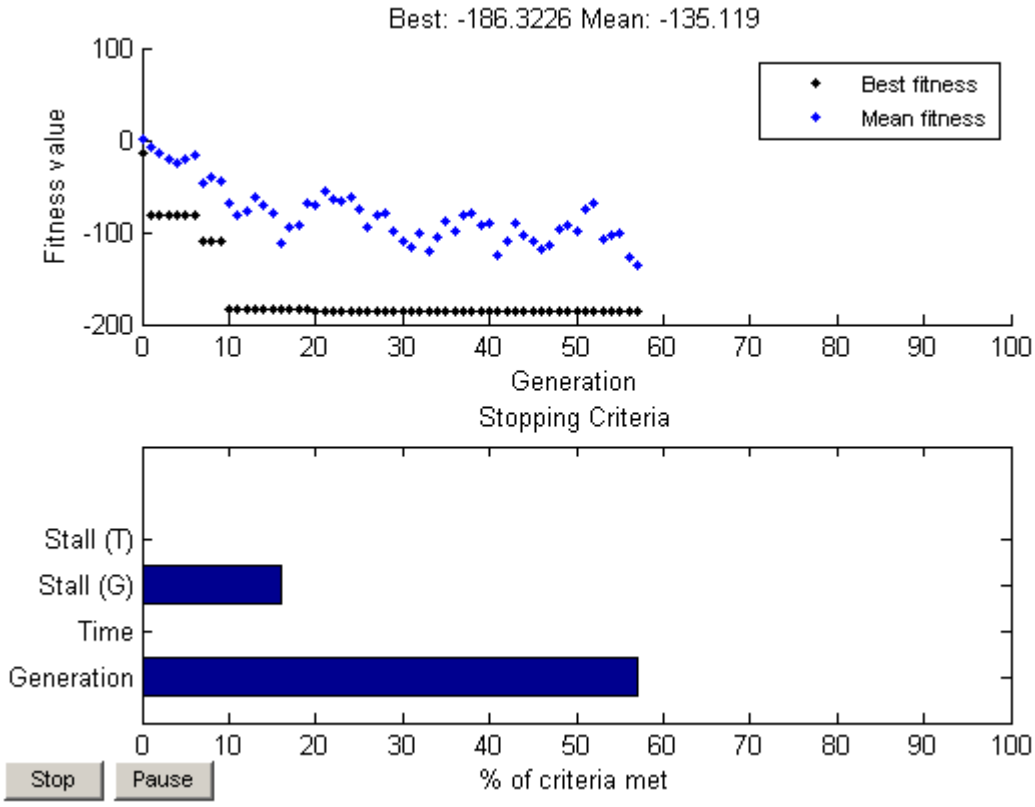
GA can accept one or more plot functions through an OPTIONS argument. This feature is useful for visualizing the performance of the solver at run time. Plot functions can be selected using GAOPTIMSET. The help for GAOPTIMSET contains a list of plot functions to choose from.

Here we use GAOPTIMSET to create an options structure to select two plot functions. The first plot function is GAPLOTBESTF, which plots the best and mean score of the population at every generation. The second plot function is GAPLOTSTOPPING, which plots the percentage of stopping criteria satisfied.

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

Run the GA solver.

```
[x,Fval,exitFlag,Output] =
ga(FitnessFunction,numberOfVariables,opts);
Optimization terminated: average change in the fitness value
less than options.TolFun.
```



Specifying Population Options

The default initial population is created using a uniform random number generator. Default values for the population size and the range of the initial population are used to create the initial population.

Specify a population size

The default population size used by GA is 20. This may not be sufficient for problems with a large number of variables; a smaller population size may be sufficient for smaller problems. Since we only have two variables, we specify a population size of 10. We will pass our options structure 'opts', created above, to GAOPTIMSET to modify the value of the parameter 'PopulationSize' to be 10.

```
opts = gaoptimset(opts,'PopulationSize',10);
```

Specify initial population range

The initial population is generated using a uniform random number generator in a default range of [0;1]. This creates an initial population where all the points are in the range 0 to 1. For example, a population of size 3 in a problem with two variables could look like:

```
Population = rand(3,2)
Population =

    0.1481    0.8734
    0.4835    0.6256
    0.2772    0.5116
```

The initial range can be set by changing the 'PopInitRange' option using GAOPTIMSET. The range must be a matrix with two rows. If the range has only one column, i.e., it is 2-by-1, then the range of every variable is the given range. For example, if we set the range to [-1; 1], then the initial range for both our variables is -1 to 1. 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).

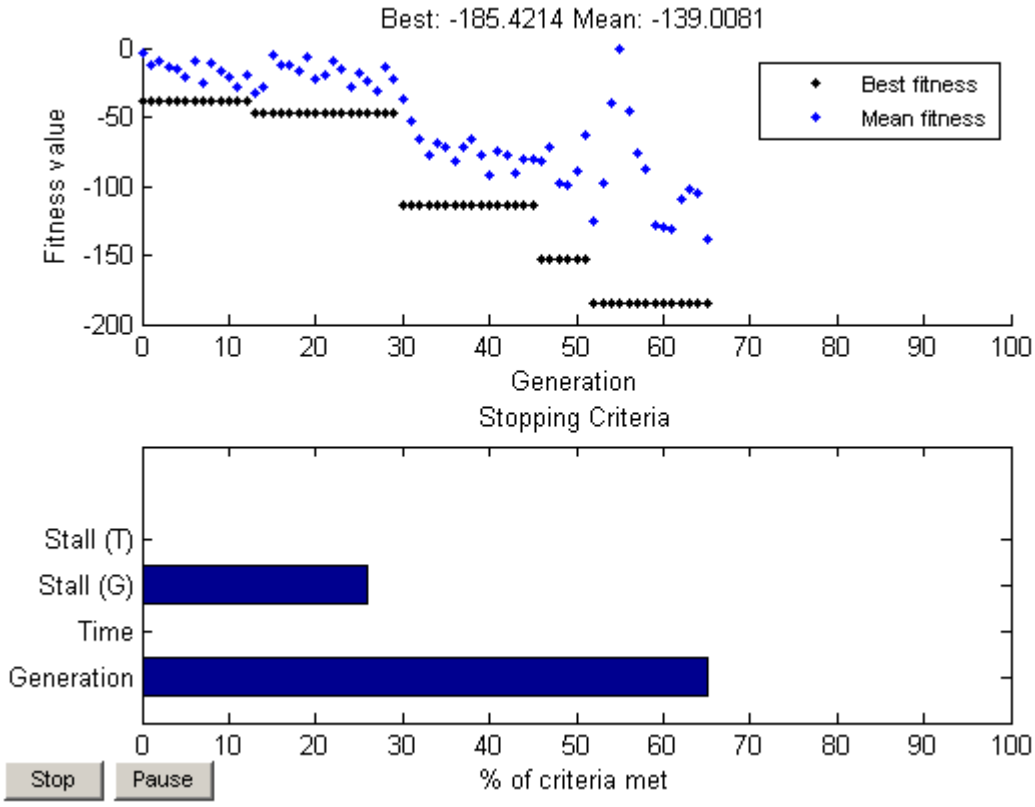
The initial range can be specified using GAOPTIMSET. We will pass our options structure 'opts' created above to GAOPTIMSET to modify the value of the parameter 'PopInitRange'.

```
opts = gaoptimset(opts,'PopInitRange',[-1 0;1 2]);
```

Run the GA 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);
Optimization terminated: average change in the fitness value
less than options.TolFun.
The number of generations was : 65
The number of function evaluations was : 660
The best function value found was : -185.421
```



Modifying the Stopping Criteria

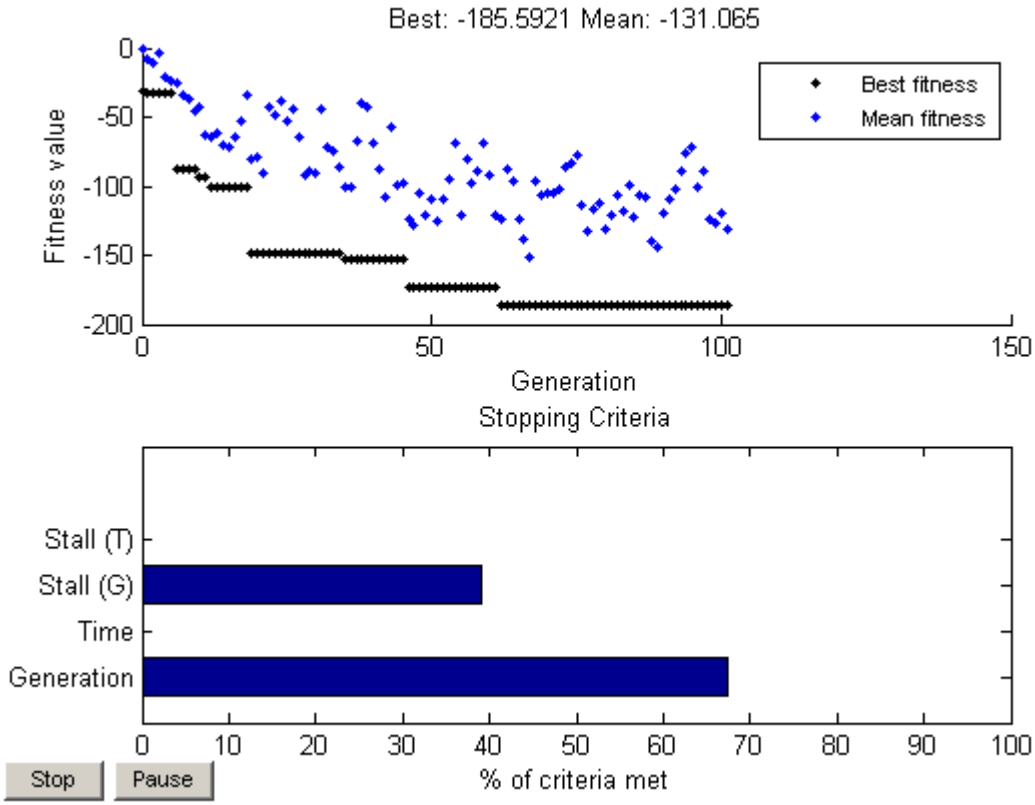
GA uses four different criteria to determine when to stop the solver. GA stops when the maximum number of generations is reached; by default this number is 100. GA also detects if there is no change in the best fitness value for some time given in seconds (stall time limit), or for some number of generations (stall generation limit). 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 limit to 100.

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

Run the GA solver again.

```
[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);
Optimization terminated: average change in the fitness value
less than options.TolFun.
The number of generations was : 101
The number of function evaluations was : 1020
The best function value found was : -185.592
```



Choosing GA Operators

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

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

Run the GA 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);
Optimization terminated: average change in the fitness value
less than options.TolFun.
The number of generations was : 51
The number of function evaluations was : 1040
The best function value found was : -179.02
```

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 OPTIMTOOL provides a wonderful environment for easily experimenting with different options and then trying them out by running the GA solver.