Genetic Algorithm and Direct Search Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 1

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Genetic Algorithm and Direct Search Toolbox User's Guide

Printing History:	January 2004	Online only	New for Version 1.0 (Release 13SP1+)
	June 2004	First printing	Revised for Version 1.0.1 (Release 14)
	October 2004	Online only	Revised for Version 1.0.2 (Release 14SP1)

Acknowledgment

The MathWorks would like to acknowledge Mark A. Abramson for his contributions to the Genetic Algorithm and Direct Search Toolbox. He researched and helped with the development of the linearly constrained pattern search algorithm.

Dr. Mark A. Abramson is the Deputy Head and Assistant Professor in the Department of Mathematics and Statistics, Air Force Institute of Technology, Wright-Patterson AFB, Ohio. Dr. Abramson is actively engaged in research for pattern search methods. He has published over 26 technical papers and has worked on NOMAD, a software package for pattern search.

Contents

1 Introducing the Genetic Algorithm and Direct Search Toolbox

What Is the Genetic Algorithm and Direct Search Toolbox? **1-2**

Writing M-Files for Functions You Want to Optimize **1-3**
Example — Writing an M-File **1-3**
Maximizing Versus Minimizing **1-4**

2 Getting Started with the Genetic Algorithm

What Is the Genetic Algorithm? **2-2**

Using the Genetic Algorithm **2-3**
Calling the Function ga at the Command Line **2-3**
Using the Genetic Algorithm Tool **2-4**

Example: Rastrigin's Function **2-6**
Rastrigin's Function **2-6**
Finding the Minimum of Rastrigin's Function **2-8**
Finding the Minimum from the Command Line **2-10**
Displaying Plots **2-11**

Some Genetic Algorithm Terminology **2-15**
Fitness Functions **2-15**
Individuals **2-15**
Populations and Generations **2-15**
Diversity **2-16**
Fitness Values and Best Fitness Values **2-16**
Parents and Children **2-17**

How the Genetic Algorithm Works **2-18**

Outline of the Algorithm 2-18
Initial Population 2-19
Creating the Next Generation 2-20
Plots of Later Generations 2-22
Stopping Conditions for the Algorithm 2-23

3 Getting Started with Direct Search

What Is Direct Search? 3-2

Performing a Pattern Search 3-3
Calling patternsearch at the Command Line 3-3
Using the Pattern Search Tool 3-3

Example: Finding the Minimum of a Function 3-6
Objective Function 3-6
Finding the Minimum of the Function 3-7
Plotting the Objective Function Values and Mesh Sizes 3-8

Pattern Search Terminology 3-10
Patterns 3-10
Meshes 3-11
Polling 3-12

How Pattern Search Works 3-13
Successful Polls 3-13
An Unsuccessful Poll 3-16
Displaying the Results at Each Iteration 3-17
More Iterations 3-18
Stopping Conditions for the Pattern Search 3-18

4 Using the Genetic Algorithm

Overview of the Genetic Algorithm Tool **4-2**
Opening the Genetic Algorithm Tool **4-2**
Defining a Problem in the Genetic Algorithm Tool **4-3**
Running the Genetic Algorithm **4-4**
Pausing and Stopping the Algorithm **4-6**
Displaying Plots **4-7**
Example — Creating a Custom Plot Function **4-8**
Reproducing Your Results **4-11**
Setting Options in the Genetic Algorithm Tool **4-12**
Importing and Exporting Options and Problems **4-13**
Example — Resuming the Genetic Algorithm from the Final Population **4-16**
Generating an M-File **4-20**

Using the Genetic Algorithm from the Command Line . . . **4-21**
Running ga with the Default Options **4-21**
Setting Options for ga at the Command Line **4-22**
Using Options and Problems from the Genetic Algorithm Tool **4-24**
Reproducing Your Results **4-25**
Resuming ga from the Final Population of a Previous Run . . **4-26**
Running ga from an M-File **4-27**

Genetic Algorithm Examples **4-30**
Population Diversity **4-30**
Fitness Scaling **4-35**
Selection **4-39**
Reproduction Options **4-40**
Mutation and Crossover **4-40**
Setting the Amount of Mutation **4-41**
Setting the Crossover Fraction **4-43**
Comparing Results for Varying Crossover Fractions **4-46**
Example — Global Versus Local Minima **4-48**
Using a Hybrid Function **4-52**
Setting the Maximum Number of Generations **4-54**
Vectorizing the Fitness Function **4-56**

5 Using Direct Search

Overview of the Pattern Search Tool **5-2**
Opening the Pattern Search Tool **5-2**
Defining a Problem in the Pattern Search Tool **5-3**
Running a Pattern Search **5-5**
Example — A Constrained Problem **5-6**
Pausing and Stopping the Algorithm **5-8**
Displaying Plots **5-8**
Setting Options in the Pattern Search Tool **5-10**
Importing and Exporting Options and Problems **5-11**
Generating an M-File **5-13**

Performing a Pattern Search from the Command Line . . **5-14**
Calling patternsearch with the Default Options **5-14**
Setting Options for patternsearch at the Command Line **5-16**
Using Options and Problems from the Pattern Search Tool . . **5-18**

Pattern Search Examples **5-19**
Poll Method **5-19**
Complete Poll **5-21**
Using a Search Method **5-25**
Mesh Expansion and Contraction **5-28**
Mesh Accelerator **5-33**
Using Cache **5-34**
Setting Tolerances for the Solver **5-36**

Parameterizing Functions Called by patternsearch or ga **5-42**
Parameterizing Functions Using Anonymous Functions **5-42**
Parameterizing a Function Using a Nested Function **5-44**

6 Function Reference

Functions — Categorical List **6-2**
Genetic Algorithm **6-2**

Direct Search 6-2

Genetic Algorithm Options 6-3
Plot Options 6-4
Population Options 6-6
Fitness Scaling Options 6-8
Selection Options 6-9
Reproduction Options 6-11
Mutation Options 6-11
Crossover Options 6-14
Migration Options 6-16
Hybrid Function Option 6-17
Stopping Criteria Options 6-18
Output Function Options 6-18
Display to Command Window Options 6-19
Vectorize Option 6-20

Pattern Search Options 6-21
Plot Options 6-22
Poll Options 6-24
Search Options 6-26
Mesh Options 6-30
Cache Options 6-31
Stopping Criteria 6-31
Output Function Options 6-32
Display to Command Window Options 6-34
Vectorize Option 6-34

Functions — Alphabetical List 6-36

Index

1

Introducing the Genetic Algorithm and Direct Search Toolbox

What Is the Genetic Algorithm and Direct Search Toolbox? (p. 1-2)	Introduces the toolbox and its features.
Writing M-Files for Functions You Want to Optimize (p. 1-3)	Explains how to write M-files that compute the functions you want to optimize.

What Is the 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 the MATLAB® 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 standard Optimization Toolbox.

All the toolbox functions are MATLAB M-files, made up of MATLAB statements that implement specialized optimization algorithms. You can view the MATLAB code for these functions using the statement

```
type function_name
```

You can extend the capabilities of the Genetic Algorithm and Direct Search Toolbox by writing your own M-files, or by using the toolbox in combination with other toolboxes, or with MATLAB or Simulink®.

Writing M-Files for Functions You Want to Optimize

To use the Genetic Algorithm and Direct Search Toolbox, you must first write an M-file that computes the function you want to optimize. The M-file should accept a row vector, whose length is the number of independent variables for the objective function, and return a scalar. This section explains how to write the M-file and covers the following topics:

- "Example — Writing an M-File" on page 1-3
- "Maximizing Versus Minimizing" on page 1-4

Example — Writing an M-File

The following example shows how to write an M-file for the function you want to optimize. Suppose that you want to minimize the function

$$f(x_1, x_2) = x_1^2 - 2x_1x_2 + 6x_1 + x_2^2 - 6x_2$$

The M-file that computes this function must accept a row vector `x` of length 2, corresponding to the variables x_1 and x_2, and return a scalar equal to the value of the function at `x`. To write the M-file, do the following steps:

1. Select **New** in the MATLAB **File** menu.
2. Select **M-File**. This opens a new M-file in the editor.
3. In the M-file, enter the following two lines of code:

```
function z = my_fun(x)
z = x(1)^2 - 2*x(1)*x(2) + 6*x(1) + x(2)^2 - 6*x(2);
```

4. Save the M-file in a directory on the MATLAB path.

To check that the M-file returns the correct value, enter

```
my_fun([2 3])

ans =

    -5
```

Note Do not use the Editor/Debugger to debug the M-file for the objective function while running the Genetic Algorithm Tool or the Pattern Search Tool. Doing so results in Java exception messages in the Command Window and makes debugging more difficult. See either "Defining a Problem in the Genetic Algorithm Tool" on page 4-3 or "Defining a Problem in the Pattern Search Tool" on page 5-3 for more information on debugging.

Maximizing Versus Minimizing

The optimization functions in the Genetic Algorithm and Direct Search Toolbox minimize the objective or fitness function. That is, they solve problems of the form

$$\underset{x}{\text{minimize}} \quad f(x)$$

If you want to maximize $f(x)$, you can do so by minimizing $-f(x)$, because the point at which the minimum of $-f(x)$ occurs is the same as the point at which the maximum of $f(x)$ occurs.

For example, suppose you want to maximize the function

$$f(x_1, x_2) = x_1^2 - 2x_1x_2 + 6x_1 + x_2^2 - 6x_2$$

described in the preceding section. In this case, you should write your M-file to compute

$$-f(x_1, x_2) = -x_1^2 + 2x_1x_2 - 6x_1 - x_2^2 + 6x_2$$

and minimize this function.

2

Getting Started with the Genetic Algorithm

What Is the Genetic Algorithm? (p. 2-2)	Introduces the genetic algorithm.
Using the Genetic Algorithm (p. 2-3)	Explains how to use the genetic algorithm tool.
Example: Rastrigin's Function (p. 2-6)	Presents an example of solving an optimization problem using the genetic algorithm.
Some Genetic Algorithm Terminology (p. 2-15)	Explains some basic terminology for the genetic algorithm.
How the Genetic Algorithm Works (p. 2-18)	Presents an overview of how the genetic algorithm works.

What Is the Genetic Algorithm?

The genetic algorithm is a method for solving optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear.

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 differs from a standard optimization algorithm in two main ways, as summarized in the following table.

Standard Algorithm	Genetic Algorithm
Generates a single point at each iteration. The sequence of points approaches an optimal solution.	Generates a population of points at each iteration. The population approaches an optimal solution.
Selects the next point in the sequence by a deterministic computation.	Selects the next population by computations that involve random choices.

Using the Genetic Algorithm

There are two ways you can use the genetic algorithm with the toolbox:

- Calling the genetic algorithm function `ga` at the command line.
- Using the Genetic Algorithm Tool, a graphical interface to the genetic algorithm.

This section provides a brief introduction to these methods.

Calling the Function ga at the Command Line

To use 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

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

Using the function `ga` is convenient if you want to

- Return results directly to the MATLAB workspace
- Run the genetic algorithm multiple times with different options, by calling `ga` from an M-file

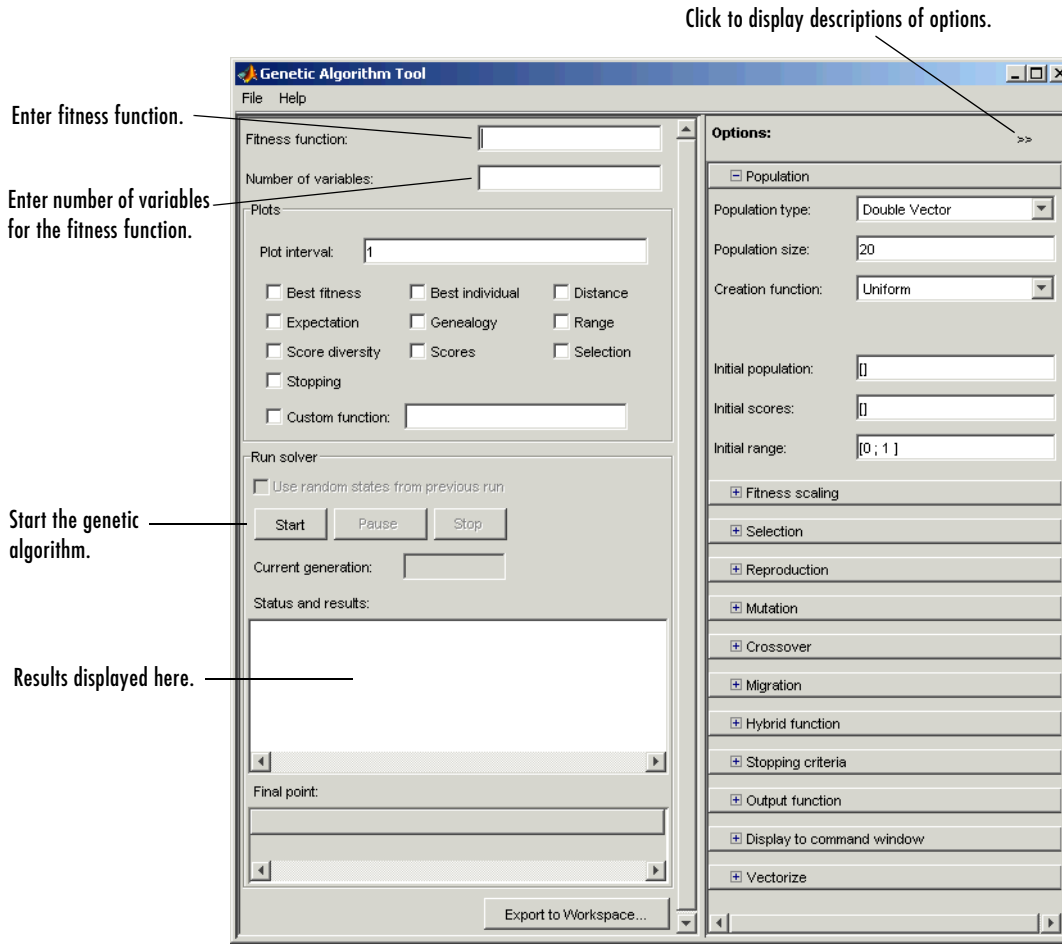
"Using the Genetic Algorithm from the Command Line" on page 4-21 provides a detailed description of using the function `ga` and creating the options structure.

Using the Genetic Algorithm Tool

The Genetic Algorithm Tool is a graphical user interface that enables you to use the genetic algorithm without working at the command line. To open the Genetic Algorithm Tool, enter

```
gatool
```

This opens the tool as shown in the following figure.



To use the Genetic Algorithm 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. "Writing M-Files for Functions You Want to Optimize" on page 1-3 explains how write this M-file. The `@` sign creates a function handle to `fitnessfun`.
- **Number of variables** — The length of the input vector to the fitness function. For the function `my_fun` described in "Writing M-Files for Functions You Want to Optimize" on page 1-3, you would enter `2`.

To run the genetic algorithm, click the **Start** button. The tool displays the results of the optimization in the **Status and 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.

For more information,

- See "Overview of the Genetic Algorithm Tool" on page 4-2 for a detailed description of the tool.
- See "Example: Rastrigin's Function" on page 2-6 for an example of using the tool.

Example: Rastrigin's Function

This section presents an example that shows how to find the minimum of Rastrigin's function, a function that is often used to test the genetic algorithm. This section covers the following topics:

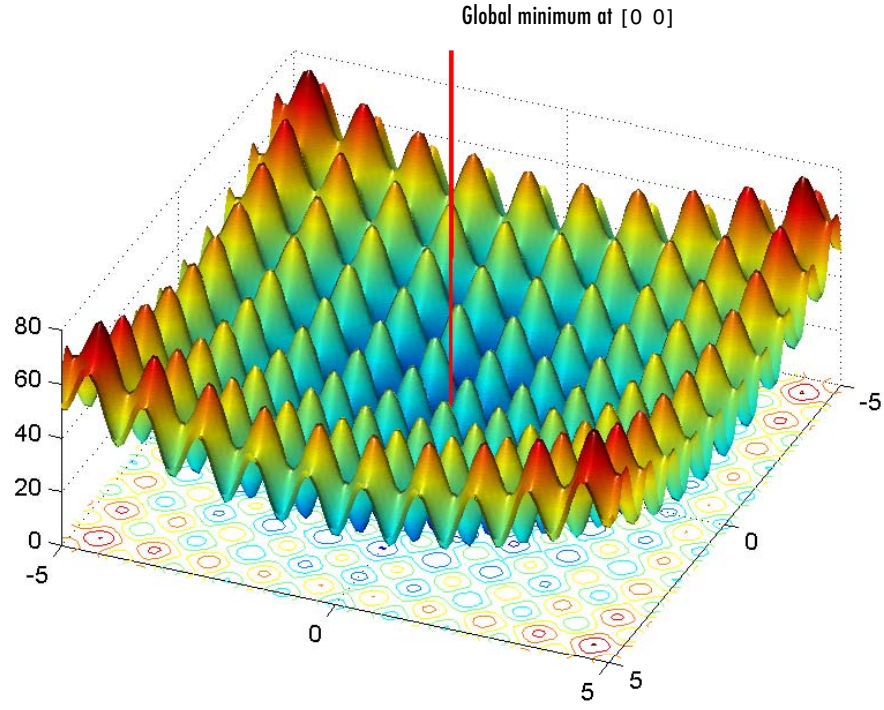
- "Rastrigin's Function" on page 2-6
- "Finding the Minimum of Rastrigin's Function" on page 2-8
- "Displaying Plots" on page 2-11

Rastrigin's Function

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

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

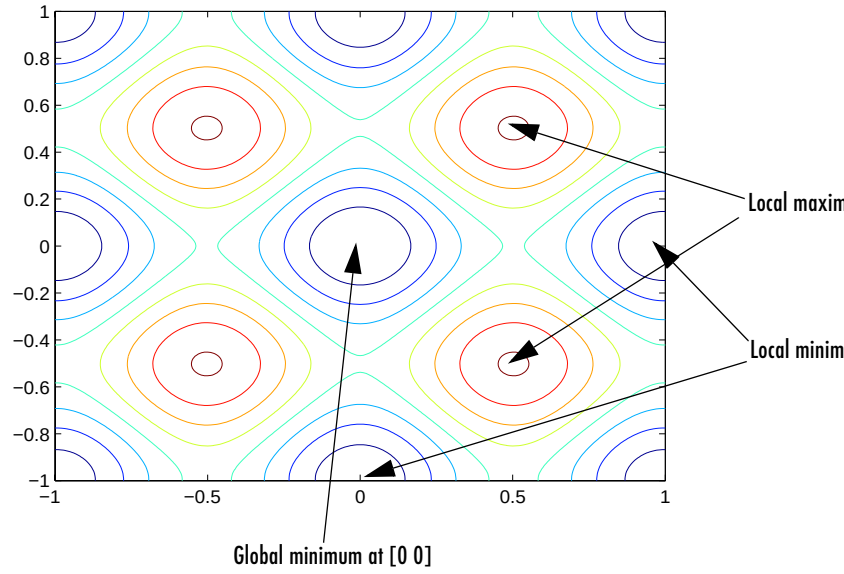
The toolbox contains an M-file, `rastriginsfcn.m`, that computes the values of Rastrigin's function. The following figure shows a plot of Rastrigin's function.



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.



Finding the Minimum of Rastrigin's Function

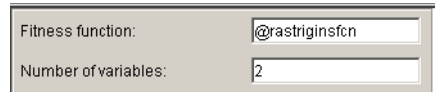
This section explains how to find the minimum of Rastrigin's function using the genetic algorithm.

Note Because the genetic algorithm uses random data to perform its search, the algorithm returns slightly different results each time you run it.

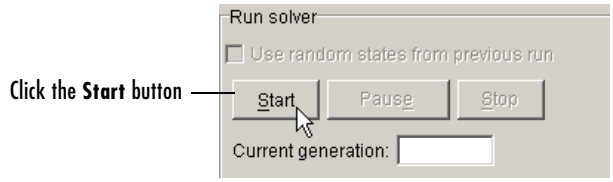
To find the minimum, do the following steps:

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.

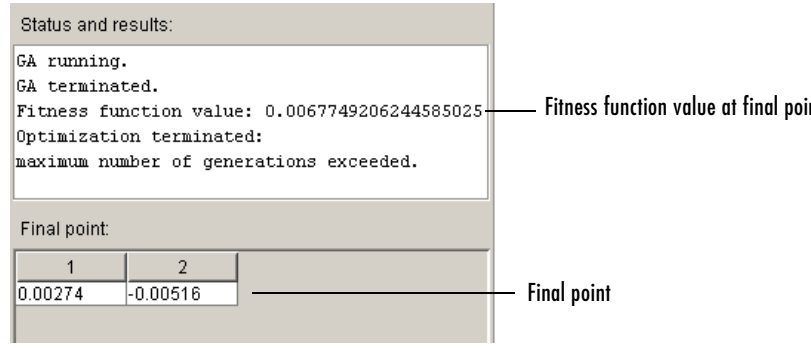


3 Click the **Start** button in the **Run solver** pane, as shown in the following figure.



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: 0.0067749206244585025
  ```

  Note that the value shown is very close to the actual minimum value of Rastrigin's function, which is `0`. "Genetic Algorithm Examples" on page 4-30 describes some ways to get a result that is closer to the actual minimum.
- The reason the algorithm terminated.

  ```
  Exit: Optimization terminated:
  maximum number of generations exceeded.
  ```

  In this example, the algorithm terminates after 100 generations, the default value of the option **Generations**, which specifies the maximum number of generations the algorithm computes.
- The final point, which in this example is `[0.00274 -0.00516]`.

Finding the Minimum from the Command Line

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

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

This returns

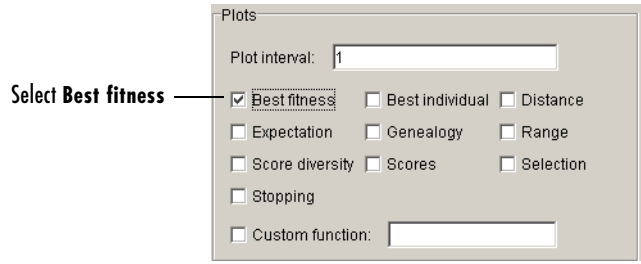
```
[x fval reason] = ga(@rastriginsfcn, 2)

x =

    0.0027   -0.0052


fval =

    0.0068


reason =

Optimization terminated:
maximum number of generations exceeded.
```

where

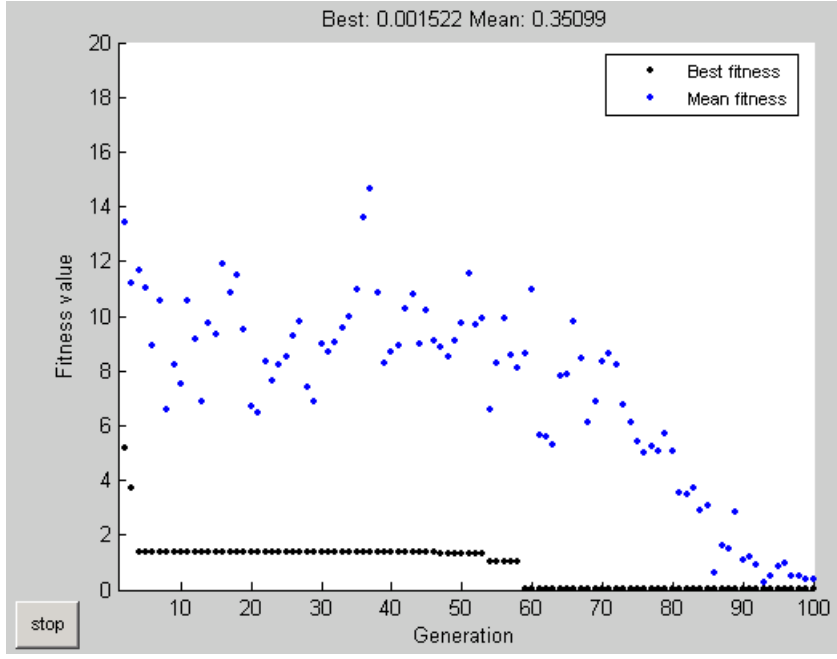
- `x` is the final point returned by the algorithm.
- `fval` is the fitness function value at the final point.
- `reason` is the reason that the algorithm terminated.

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 value**, as shown in the following figure.



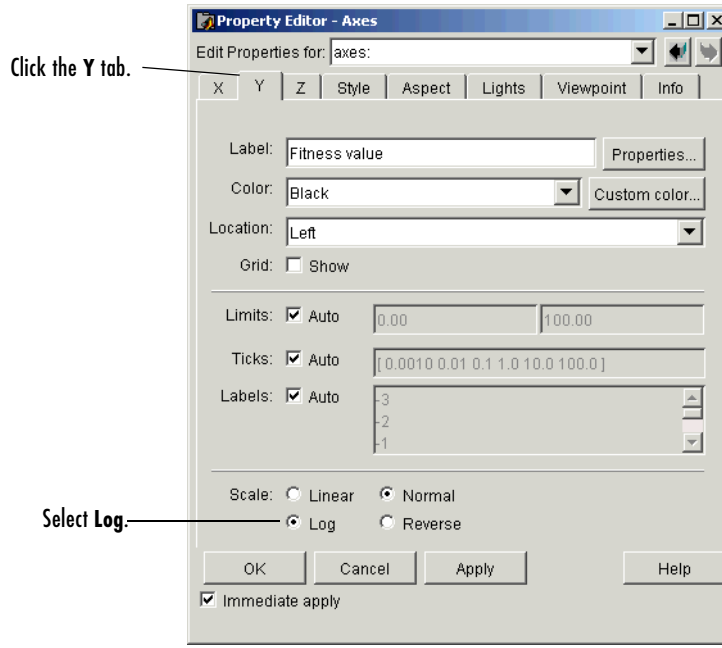
When you click **Start**, the Genetic Algorithm Tool displays a plot of the best and mean values of the fitness function at each generation. When the algorithm stops, the plot appears as shown in the following figure.



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,

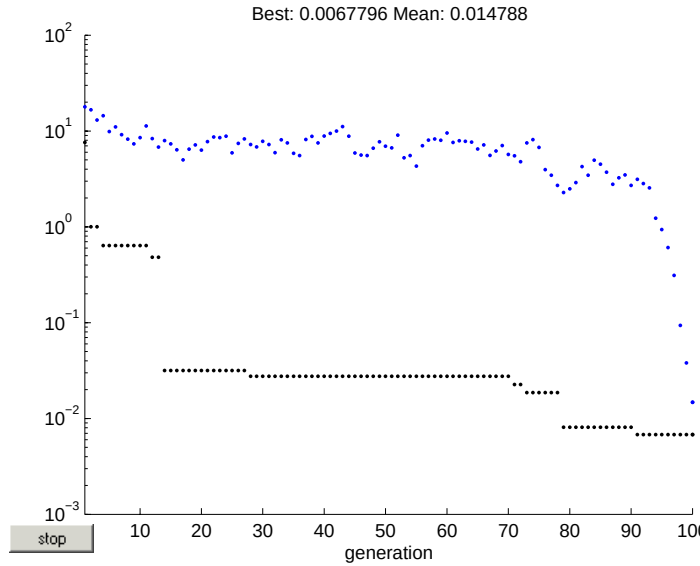
1 Select **Axes Properties** from the **Edit** menu in the plot window to open the Property Editor, as shown in the following figure.



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.

Some Genetic Algorithm Terminology

This section explains some basic terminology for the genetic algorithm, including

- "Fitness Functions" on page 2-15
- "Individuals" on page 2-15
- "Populations and Generations" on page 2-15
- "Fitness Values and Best Fitness Values" on page 2-16
- "Parents and Children" on page 2-17

Fitness Functions

The *fitness function* is the function you want to optimize. For standard optimization algorithms, this is known as the objective function. The toolbox tries to find the minimum of the fitness function.

You can write the fitness function as an M-file and pass it as an input argument to the main genetic algorithm function.

Individuals

An *individual* is any point to which you can apply the fitness function. The value of the fitness function for an individual is its score. For example, if the fitness function is

$$f(x_1, x_2, x_3) = (2x_1 + 1)^2 + (3x_2 + 4)^2 + (x_3 - 2)^2$$

the vector (2, 3, 1), whose length is the number of variables in the problem, is an individual. The score of the individual (2, 3, 1) is $f(2, -3, 1) = 51$.

An individual is sometimes referred to as a *genome* and the vector entries of an individual as *genes*.

Populations and Generations

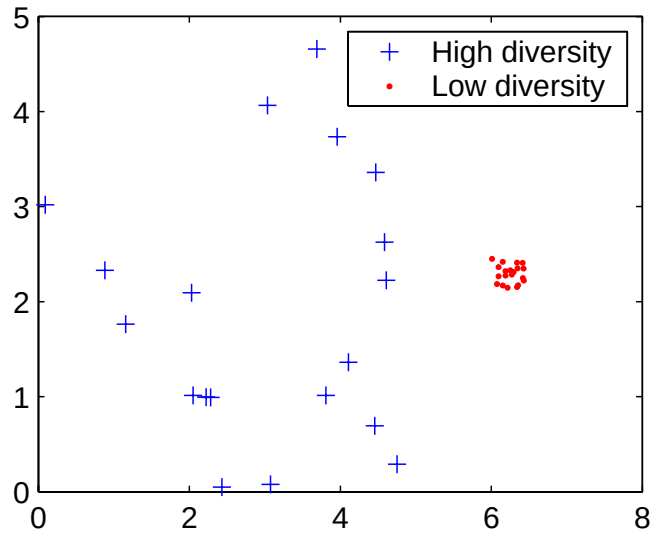
A *population* is an array of individuals. For example, if the size of the population is 100 and the number of variables in the fitness function is 3, you represent the population by a 100-by-3 matrix. The same individual can appear

more than once in the population. For example, the individual (2, 3, 1) can appear in more than one row of the array.

At each iteration, the genetic algorithm performs a series of computations on the current population to produce a new population. Each successive population is called a new *generation*.

Diversity

Diversity refers to the average distance between individuals in a population. A population has high diversity if the average distance is large; otherwise it has low diversity. In the following figure, the population on the left has high diversity, while the population on the right has low diversity.



Diversity is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

Fitness Values and Best Fitness Values

The *fitness value* of an individual is the value of the fitness function for that individual. Because the toolbox finds the minimum of the fitness function, the

best fitness value for a population is the smallest fitness value for any individual in the population.

Parents and Children

To create the next generation, the genetic algorithm selects certain individuals in the current population, called *parents*, and uses them to create individuals in the next generation, called *children*. Typically, the algorithm is more likely to select parents that have better fitness values.

How the Genetic Algorithm Works

This section provides an overview of how the genetic algorithm works. This section covers the following topics:

- "Outline of the Algorithm" on page 2-18
- "Initial Population" on page 2-19
- "Creating the Next Generation" on page 2-20
- "Plots of Later Generations" on page 2-22
- "Stopping Conditions for the Algorithm" on page 2-23

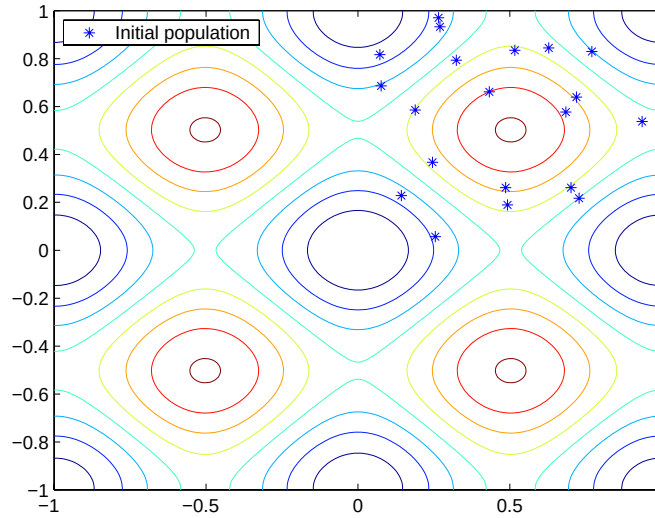
Outline of the Algorithm

The following outline summarizes how the genetic algorithm works:

1. The algorithm begins by creating a random initial population.
2. The algorithm then creates a sequence of new populations, or generations. At each step, the algorithm uses the individuals in the current generation to create the next generation. To create the new generation, the algorithm performs the following steps:
   a. Scores each member of the current population by computing its fitness value.
   b. Scales the raw fitness scores to convert them into a more usable range of values.
   c. Selects parents based on their fitness.
   d. Produces children from the parents. Children are produced either by making random changes to a single parent — *mutation* — or by combining the vector entries of a pair of parents — *crossover*.
   e. Replaces the current population with the children to form the next generation.
3. The algorithm stops when one of the stopping criteria is met. See "Stopping Conditions for the Algorithm" on page 2-23.

Initial Population

The algorithm begins by creating a random initial population, as shown in the following figure.



In this example, the initial population contains 20 individuals, which is the default value of **Population size** in the **Population** options. Note that all the individuals in the initial population lie in the upper-right quadrant of the picture, that is, their coordinates lie between 0 and 1, because the default value of **Initial range** in the **Population** options is `[0;1]`.

If you know approximately where the minimal point for a function lies, you should set **Initial range** so that the point lies near the middle of that range. For example, if you believe that the minimal point for Rastrigin's function is near the point `[0 0]`, you could set **Initial range** to be `[-1;1]`. However, as this example shows, the genetic algorithm can find the minimum even with a less than optimal choice for **Initial range**.

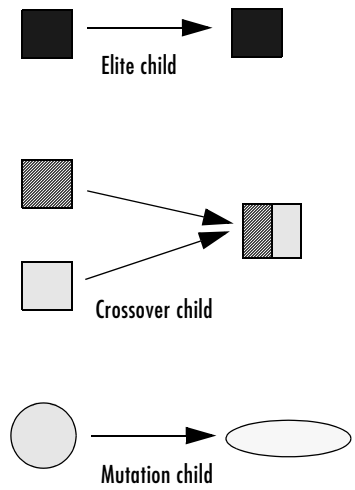
Creating the Next Generation

At each step, the genetic algorithm uses the current population to create the children that make up the next generation. The algorithm selects a group of individuals in the current population, called *parents*, who contribute their *genes* — the entries of their vectors — to their children. The algorithm usually selects individuals that have better fitness values as parents. You can specify the function that the algorithm uses to select the parents in the **Selection function** field in the **Selection** options.

The genetic algorithm creates three types of children for the next generation:

- *Elite children* are the individuals in the current generation with the best fitness values. These individuals automatically survive to the next generation.
- *Crossover children* are created by combining the vectors of a pair of parents.
- *Mutation children* are created by introducing random changes, or mutations, to a single parent.

The following schematic diagram illustrates the three types of children.



"Mutation and Crossover" on page 4-40 explains how to specify the number of children of each type that the algorithm generates and the functions it uses to perform crossover and mutation.

The following sections explain how the algorithm creates crossover and mutation children.

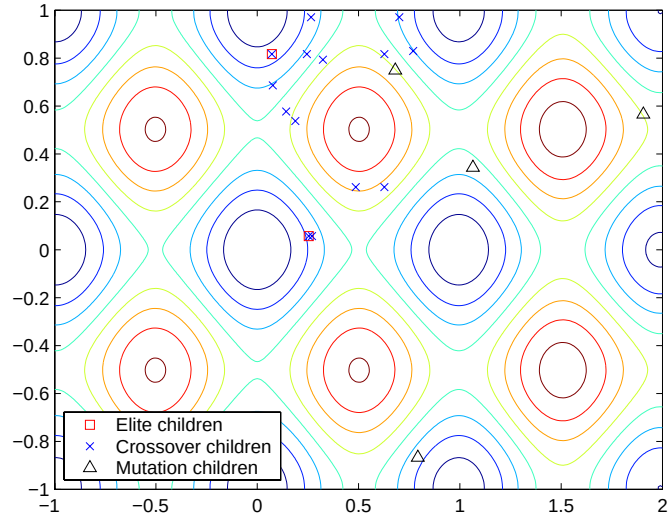
Crossover Children

The algorithm creates crossover children by combining pairs of parents in the current population. At each coordinate of the child vector, the default crossover function randomly selects an entry, or *gene*, at the same coordinate from one of the two parents and assigns it to the child.

Mutation Children

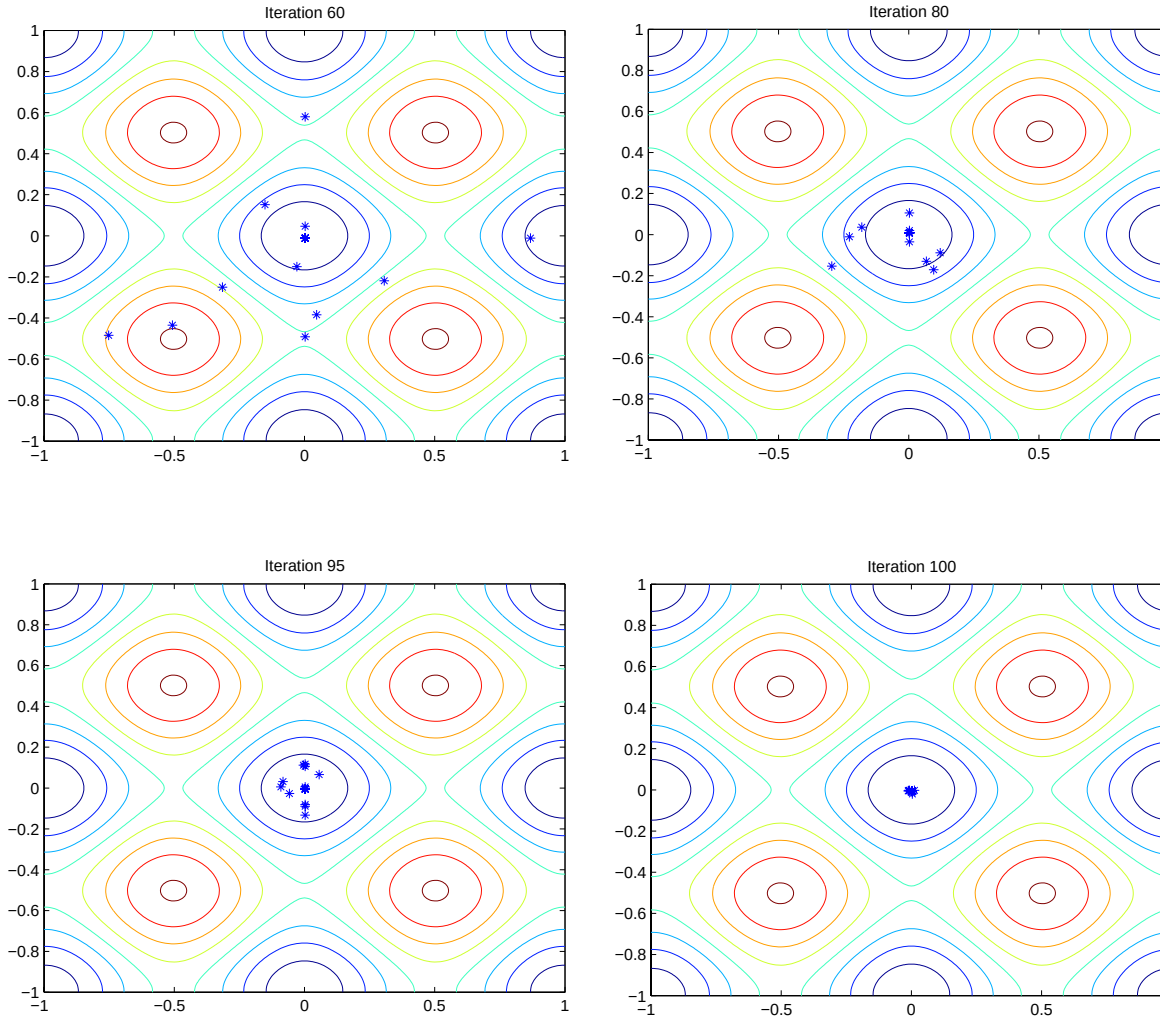
The algorithm creates mutation children by randomly changing the genes of individual parents. By default, the algorithm adds a random vector from a Gaussian distribution to the parent.

The following figure shows the children of the initial population, that is, the population at the second generation, and indicates whether they are elite, crossover, or mutation children.



Plots of Later Generations

The following figure shows the populations at iterations 60, 80, 95, and 100.



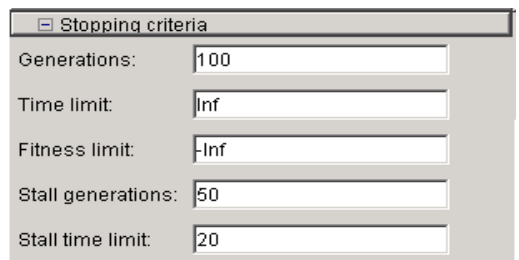
As the number of generations increases, the individuals in the population get closer together and approach the minimum point `[0 0]`.

Stopping Conditions for the Algorithm

The genetic algorithm uses the following five conditions to determine when to stop:

- **Generations** — The algorithm stops when the number of generations reaches the value of **Generations**.
- **Time limit** — The algorithm stops after running for an amount of time in seconds equal to **Time limit**.
- **Fitness limit** — The algorithm stops when the value of the fitness function for the best point in the current population is less than or equal to **Fitness limit**.
- **Stall generations** — The algorithm stops if there is no improvement in the objective function for a sequence of consecutive generations of length **Stall generations**.
- **Stall time limit** — The algorithm stops if there is no improvement in the objective function during an interval of time in seconds equal to **Stall time limit**.

The algorithm stops as soon as any one of these five conditions is met. You can specify the values of these criteria in the **Stopping criteria** options in the Genetic Algorithm Tool. The default values are shown in the figure below.



When you run the genetic algorithm, the **Status** panel displays the criterion that caused the algorithm to stop.

The options **Stall time limit** and **Time limit** prevent the algorithm from running too long. If the algorithm stops due to one of these conditions, you

might improve your results by increasing the values of **Stall time limit** and **Time limit**.

3

Getting Started with Direct Search

What Is Direct Search? (p. 3-2)	Introduces direct search and pattern search.
Performing a Pattern Search (p. 3-3)	Explains the main function in the toolbox for performing pattern search.
Example: Finding the Minimum of a Function (p. 3-6)	Provides an example of solving an optimization problem using pattern search.
Pattern Search Terminology (p. 3-10)	Explains some basic pattern search terminology.
How Pattern Search Works (p. 3-13)	Provides an overview of direct search algorithms.
Plotting the Objective Function Values and Mesh Sizes (p. 3-8)	Shows how to plot the objective function values and mesh sizes of the sequence of points generated by the pattern search.

What Is Direct Search?

Direct search is a method for solving optimization problems that does not require any information about the gradient of the objective function. As opposed to more traditional optimization methods that use information about the gradient or higher derivatives to search for an optimal point, a direct search algorithm searches a set of points around the current point, looking for one where the value of the objective function is lower than the value at the current point. You can use direct search to solve problems for which the objective function is not differentiable, or even continuous.

The Genetic Algorithm and Direct Search Toolbox implements a special class of direct search algorithms called *pattern search* algorithms. A pattern search algorithm computes a sequence of points that get closer and closer to the optimal point. At each step, the algorithm searches a set of points, called a *mesh*, around the *current point* — the point computed at the previous step of the algorithm. The algorithm forms the mesh by adding the current point to a scalar multiple of a fixed set of vectors called a *pattern*. If the algorithm finds a point in the mesh that improves the objective function at the current point, the new point becomes the current point at the next step of the algorithm.

Performing a Pattern Search

This section provides a brief introduction to the Pattern Search Tool, a graphical user interface (GUI) for performing a pattern search. This section covers the following topics:

- "Calling patternsearch at the Command Line" on page 3-3
- "Using the Pattern Search Tool" on page 3-3

Calling patternsearch at the Command Line

To perform a pattern search on an unconstrained problem at the command line, you call the function `patternsearch` with the syntax

```
[x fval] = patternsearch(@objfun, x0)
```

where

- `@objfun` is a handle to the objective function.
- `x0` is the starting point for the pattern search.

The results are given by

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

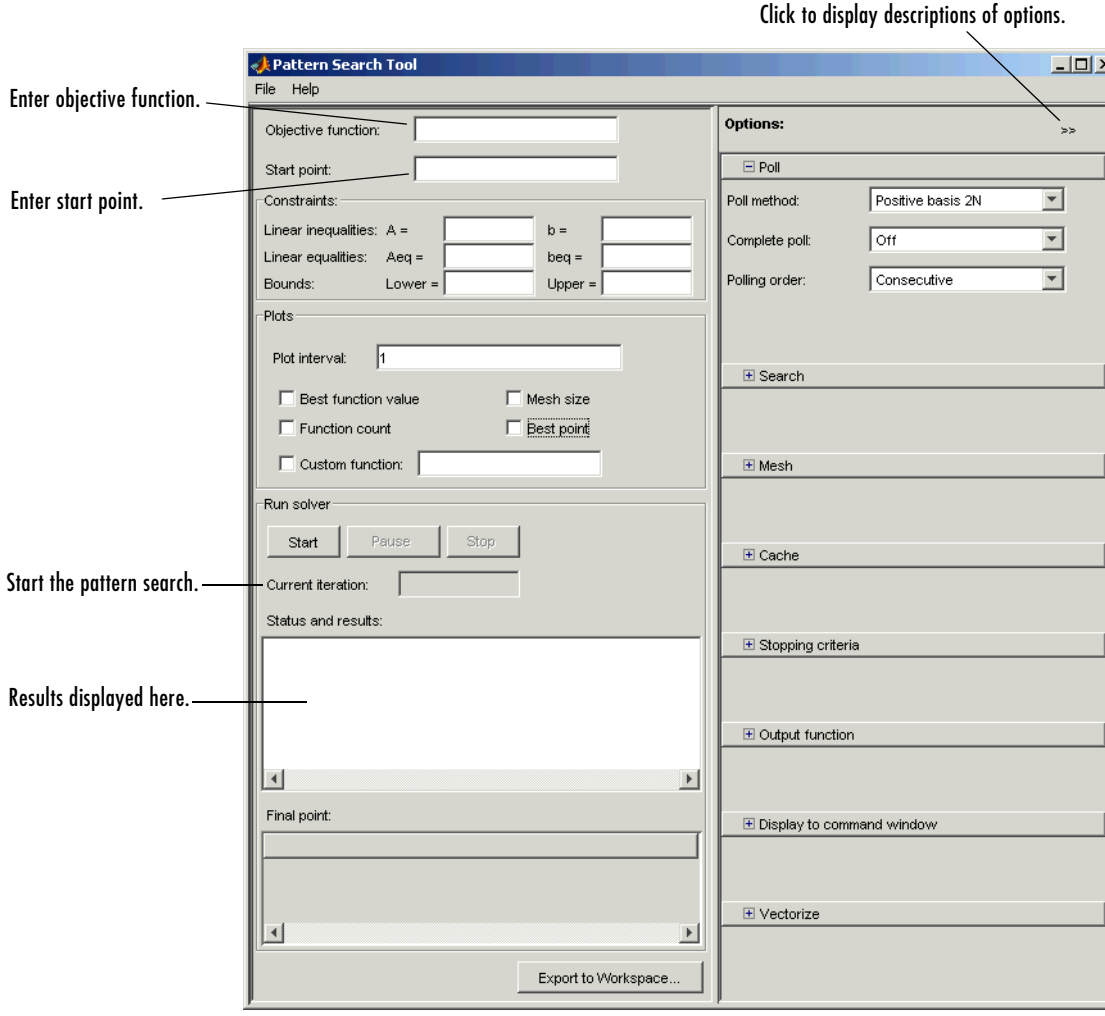
"Performing a Pattern Search from the Command Line" on page 5-14 explains in detail how to use the function `patternsearch`.

Using the Pattern Search Tool

To open the Pattern Search Tool, enter

```
psearchtool
```

This opens the tool as shown in the following figure.


Click to display descriptions of options.
Enter objective function.
Enter start point.
Start the pattern search.
Results displayed here.
Pattern Search Tool
File Help
Objective function:
Start point:
Constraints:
Linear inequalities: A =
b =
Linear equalities: Aeq =
beq =
Bounds: Lower =
Upper =
Plots
Plot interval: 1
Best function value
Mesh size
Function count
Best point
Custom function:
Run solver
Start
Pause
Stop
Current iteration:
Status and results:
Final point:
Export to Workspace...
Options:
>>
Poll
Poll method: Positive basis 2N
Complete poll: Off
Polling order: Consecutive
Search
Mesh
Cache
Stopping criteria
Output function
Display to command window
Vectorize

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

- **Objective function** — The objective function you want to minimize. You enter the objective function in the form `@objfun`, where `objfun.m` is an M-file that computes the objective function. The `@` sign creates a function handle to `objfun`.
- **Start point**— The initial point at which the algorithm starts the optimization.

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

Then, click the **Start** button. The tool displays the results of the optimization in the **Status and results** pane.

You can also change the options for the pattern search in the **Options** pane. To view the options in a category, click the **+** sign next to it.

"Finding the Minimum of the Function" on page 3-7 gives an example of using the Pattern Search Tool.

"Overview of the Pattern Search Tool" on page 5-2 provides a detailed description of the Pattern Search Tool.

Example: Finding the Minimum of a Function

This section presents an example of using a pattern search to find the minimum of a function. This section covers the following topics:

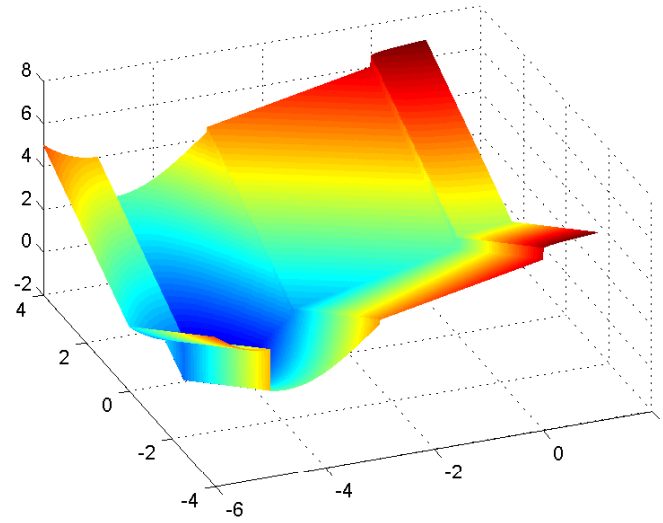
- "Objective Function" on page 3-6
- "Finding the Minimum of the Function" on page 3-7
- "Plotting the Objective Function Values and Mesh Sizes" on page 3-8

Objective Function

The example uses the objective function, `ps_example`, which is included in the Genetic Algorithms and Direct Search Toolbox. You can view the code for the function by entering

```
type ps_example
```

The following figure shows a plot of the function.



Finding the Minimum of the Function

To find the minimum of `ps_example`, do the following steps:

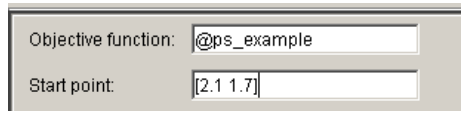
1. Enter

   ```
   psearchtool
   ```

   to open the Pattern Search Tool.

2. In the **Objective function** field of the Pattern Search Tool, enter `@ps_example`.

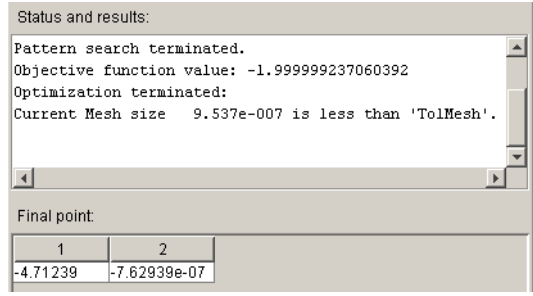
3. In the **Start point** field, type `[2.1 1.7]`.



   You can leave the fields in the **Constraints** pane blank because the problem is unconstrained.

4. Click **Start** to run the pattern search.

The **Status and Results** pane displays the results of the pattern search.

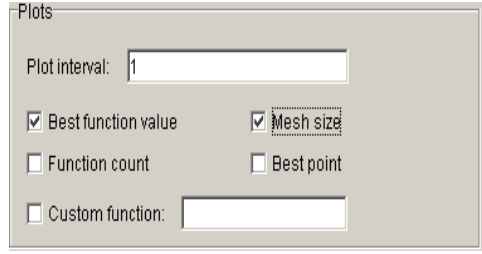


The minimum function value is approximately -2. The **Final point** pane displays the point at which the minimum occurs.

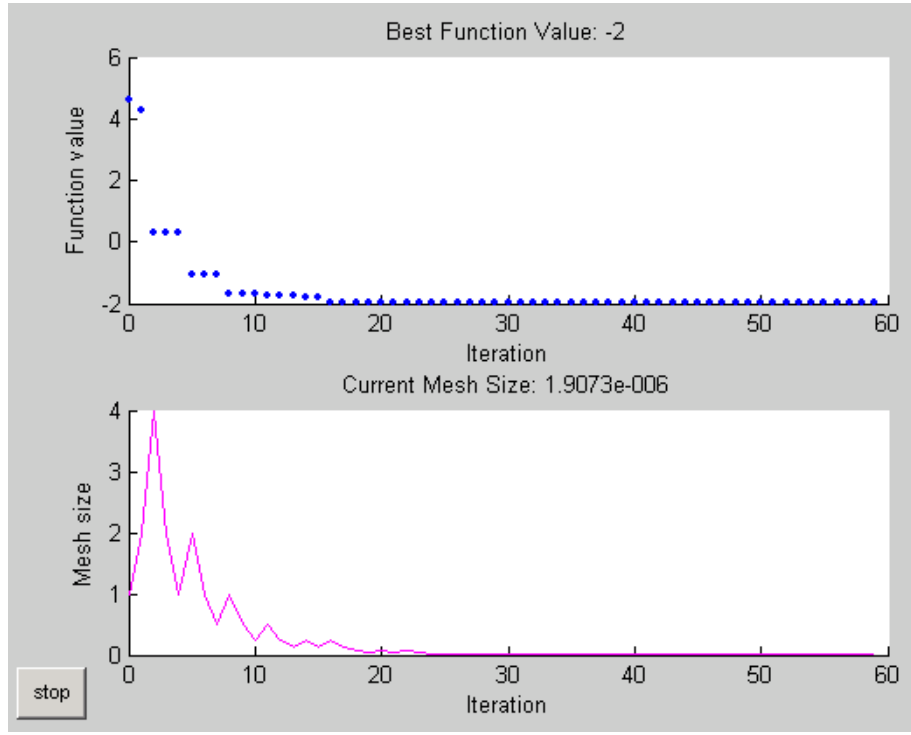
Plotting the Objective Function Values and Mesh Sizes

To see the performance of the pattern search, you can display plots of the best function value and mesh size at each iteration. First, select the following check boxes in the **Plots** pane:

- **Best function value**
- **Mesh size**



Then click **Start** to run the pattern search. This displays the following plots.



The upper plot shows the objective function value of the best point at each iteration. Typically, the objective function values improve rapidly at the early iterations and then level off as they approach the optimal value.

The lower plot shows the mesh size at each iteration. The mesh size increases after each successful iteration and decreases after each unsuccessful one, explained in “How Pattern Search Works” on page 3-13.

Pattern Search Terminology

This section explains some standard terminology for pattern search, including

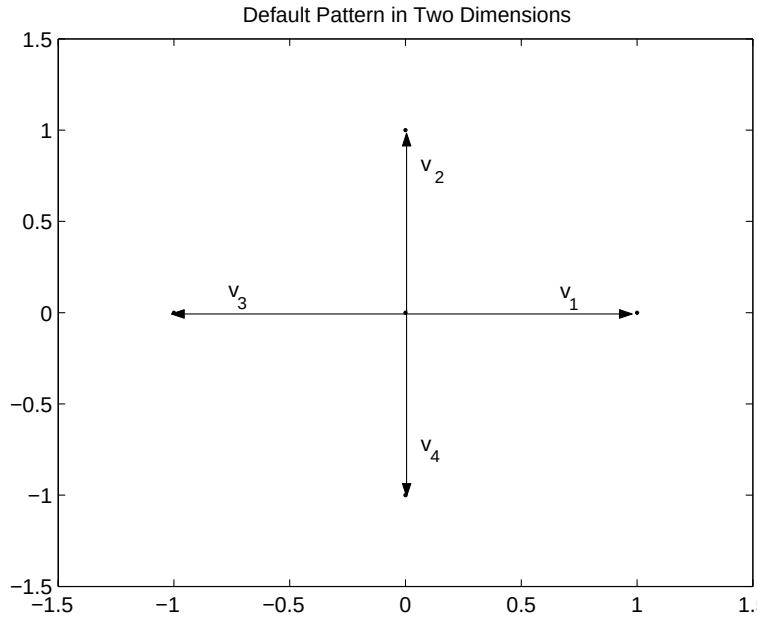
- "Patterns" on page 3-10
- "Meshes" on page 3-11
- "Polling" on page 3-12

Patterns

A *pattern* is a collection of vectors that the algorithm uses to determine which points to search at each iteration. For example, if there are two independent variables in the optimization problem, the default pattern consists of the following vectors.

v_1 = [1 0]
v_2 = [0 1]
v_3 = [-1 0]
v_4 = [0 -1]

The following figure shows these vectors.



Meshes

At each step, the pattern search algorithm searches a set of points, called a *mesh*, for a point that improves the objective function. The algorithm forms the mesh by

1. Multiplying the pattern vectors by a scalar, called the *mesh size*
2. Adding the resulting vectors to the *current point* — the point with the best objective function value found at the previous step

For example, suppose that

- The current point is `[1.6 3.4]`.

- The pattern consists of the vectors

  v_1 = [1 0]
  v_2 = [0 1]
  v_3 = [-1 0]
  v_4 = [0 -1]

- The current mesh size is `4`.

The algorithm multiplies the pattern vectors by `4` and adds them to the current point to obtain the following mesh.

```
[1.6 3.4] + 4*[1 0] = [5.6 3.4]
[1.6 3.4] + 4*[0 1] = [1.6 7.4]
[1.6 3.4] + 4*[-1 0] = [-2.4 3.4]
[1.6 3.4] + 4*[0 -1] = [1.6 -0.6]
```

The pattern vector that produces a mesh point is called its *direction*.

Polling

At each step, the algorithm polls the points in the current mesh by computing their objective function values. When option **Complete poll** has the default setting `Off`, the algorithm stops polling the mesh points as soon as it finds a point whose objective function value is less than that of the current point. If this occurs, the poll is called *successful* and the point it finds becomes the current point at the next iteration. Note that the algorithm only computes the mesh points and their objective function values up to the point at which it stops the poll. If the algorithm fails to find a point that improves the objective function, the poll is called *unsuccessful* and the current point stays the same at the next iteration.

If you set **Complete poll** to `On`, the algorithm computes the objective function values at all mesh points. The algorithm then compares the mesh point with the smallest objective function value to the current point. If that mesh point has a smaller value than the current point, the poll is successful.

How Pattern Search Works

The pattern search algorithm finds a sequence of points, `x0`, `x1`, `x2`, ... , that approaches the optimal point. The value of the objective function decreases from each point in the sequence to the next. This section explains how pattern search works for the function described in "Example: Finding the Minimum of a Function" on page 3-6.

To simplify the explanation, this section describes how the pattern search works when you set **Scale** to `Off` in **Mesh** options.

This section covers the following topics:

- "Successful Polls" on page 3-13
- "An Unsuccessful Poll" on page 3-16
- "Displaying the Results at Each Iteration" on page 3-17
- "More Iterations" on page 3-18

Successful Polls

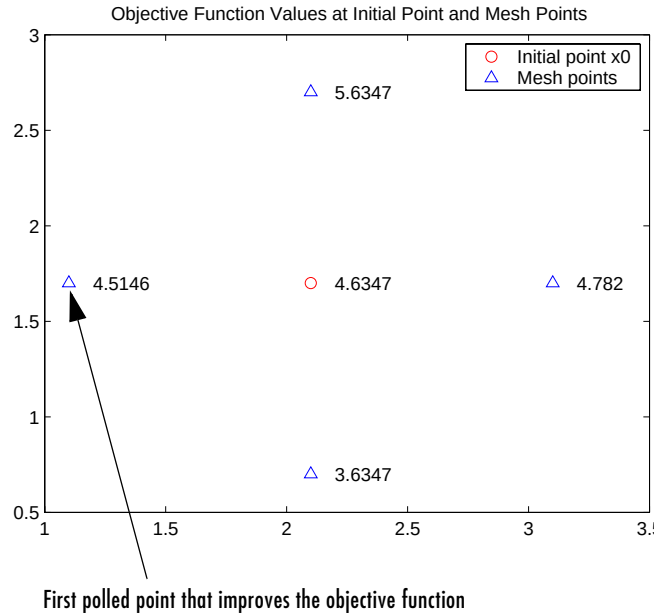
The pattern search begins at the initial point `x0` that you provide. In this example, `x0 = [2.1 1.7]`.

Iteration 1

At the first iteration, the mesh size is 1 and the pattern search algorithm adds the pattern vectors to the initial point `x0 = [2.1 1.7]` to compute the following mesh points.

```
[1 0] + x0 = [3.1 1.7]
[0 1] + x0 = [2.1 2.7]
[-1 0] + x0 = [1.1 1.7]
[0 -1 ] + x0 = [2.1 0.7]
```

The algorithm computes the objective function at the mesh points in the order shown above. The following figure shows the value of `ps_example` at the initial point and mesh points.



The algorithm polls the mesh points by computing their objective function values until it finds one whose value is smaller than 4.6347, the value at `x0`. In this case, the first such point it finds is `[1.1 1.7]`, at which the value of the objective function is `4.5146`, so the poll at iteration 1 is *successful*. The algorithm sets the next point in the sequence equal to

```
x1 = [1.1 1.7]
```

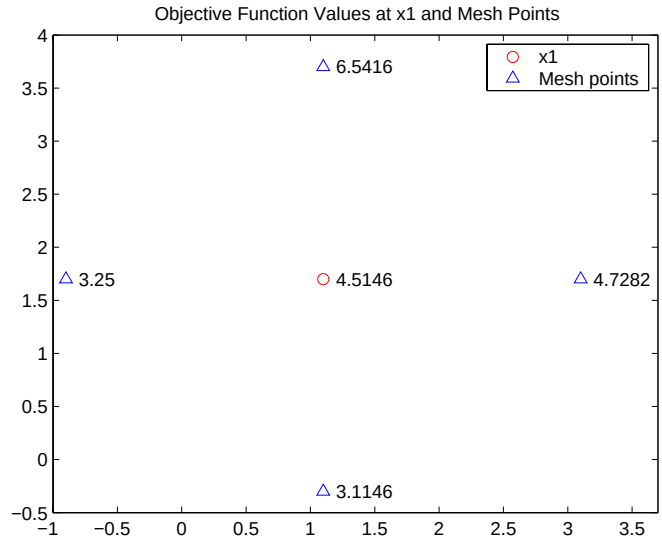
Note By default, the pattern search algorithm stops the current iteration as soon as it finds a mesh point whose fitness value is smaller than that of the current point. Consequently, the algorithm might not poll all the mesh points. You can make the algorithm poll all the mesh points by setting **Complete poll** to `On`.

Iteration 2

After a successful poll, the algorithm multiplies the current mesh size by 2, the default value of **Expansion factor** in the **Mesh** options pane. Because the initial mesh size is 1, at the second iteration the mesh size is 2. The mesh at iteration 2 contains the following points.

```
2*[1 0] + x1 = [3.1 1.7]
2*[0 1] + x1 = [1.1 3.7]
2*[-1 0] + x1 = [-0.9 1.7]
2*[0 -1 ] + x1 = [1.1 -0.3]
```

The following figure shows the point `x1` and the mesh points, together with the corresponding values of `ps_example`.



The algorithm polls the mesh points until it finds one whose value is smaller than 4.5146, the value at `x1`. The first such point it finds is `[-0.9 1.7]`, at which the value of the objective function is `3.25`, so the poll at iteration 2 is again successful. The algorithm sets the second point in the sequence equal to

```
x2 = [-0.9 1.7]
```

Because the poll is successful, the algorithm multiplies the current mesh size by 2 to get a mesh size of 4 at the third iteration.

An Unsuccessful Poll

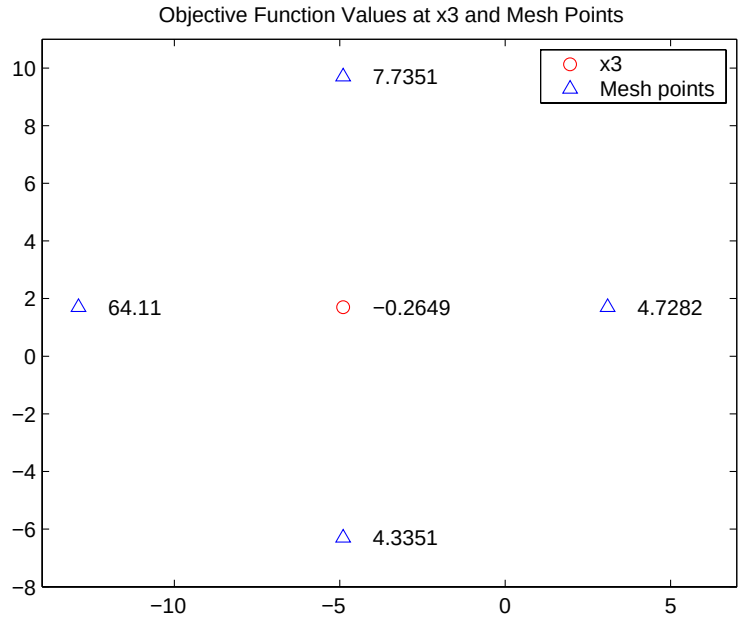
By the fourth iteration, the current point is

```
x3 = [-0.9 1.7]
```

and the mesh size is 8, so the mesh consists of the points

```
8*[1 0] + x3 = [3.1 1.7]
8*[-1 0] + x3 = [-4.9 1.7]
8*[0 1] + x3 = [-0.9 5.7]
8*[0 -1] + x3 = [-0.9 -2.3]
```

The following figure shows the mesh points and their objective function values.



At this iteration, none of the mesh points has a smaller objective function value than the value at `x3`, so the poll is *unsuccessful*. In this case, the algorithm does not change the current point at the next iteration. That is,

```
x4 = x3;
```

At the next iteration, the algorithm multiplies the current mesh size by 0.5, the default value of **Contraction factor** in the **Mesh** options pane, so that the mesh size at the next iteration is 4. The algorithm then polls with a smaller mesh size.

Displaying the Results at Each Iteration

You can display the results of the pattern search at each iteration by setting **Level of display** to **Iterative** in **Display to command window** options. This enables you to evaluate the progress of the pattern search and to make changes to `options` if necessary.

With this setting, the pattern search displays information about each iteration at the command line. The first four lines of the display are

```
Iter     f-count        MeshSize      f(x)        Method
    0        1              1         4.635     Start iterations
    1        4              2         4.515     Successful Poll
    2        7              4          3.25     Successful Poll
    3       10              8       -0.2649     Successful Poll
    4       14              4       -0.2649     Refine Mesh
```

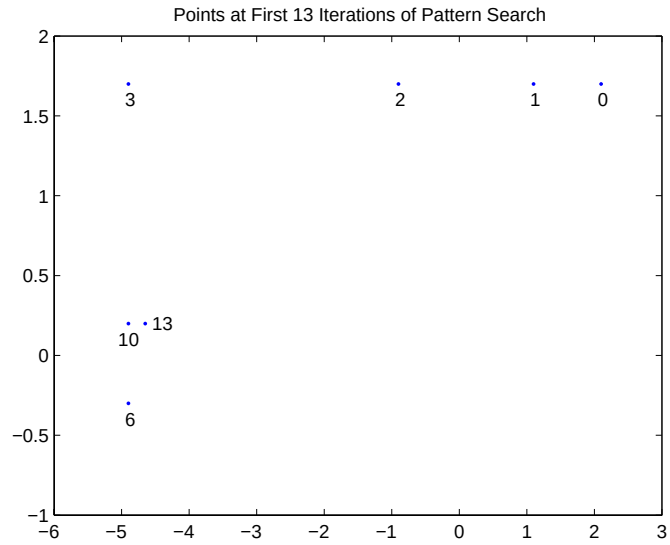
The entry `Successful Poll` below `Method` indicates that the current iteration was successful. For example, the poll at iteration 2 successful. As a result, the objective function value of the point computed at iteration 2, displayed below `f(x)`, is less than the value at iteration 1.

At iteration 4, the entry `Refine Mesh` below `Method` tells you that the poll is unsuccessful. As a result, the function value at iteration 4 remains unchanged from iteration 3.

Note that the pattern search doubles the mesh size after each successful poll and halves it after each unsuccessful poll.

More Iterations

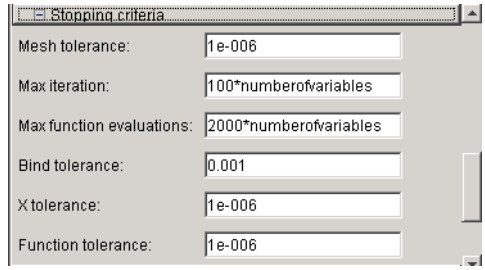
The pattern search performs 88 iterations before stopping. The following plot shows the points in the sequence computed in the first 13 iterations of the pattern search.



The numbers below the points indicate the first iteration at which the algorithm finds the point. The plot only shows iteration numbers corresponding to successful polls, because the best point doesn't change after an unsuccessful poll. For example, the best point at iterations 4 and 5 is the same as at iteration 3.

Stopping Conditions for the Pattern Search

This section describes the criteria for stopping the pattern search algorithm. These criteria are listed in the **Stopping criteria** section of the Pattern Search Tool, as shown in the following figure.



The algorithm stops when any of the following conditions occurs:

- The mesh size is less than **Mesh tolerance**.
- The number of iterations performed by the algorithm reaches the value of **Max iteration**.
- The total number of objective function evaluations performed by the algorithm reaches the value of **Max function evaluations**.
- The distance between the point found at one successful poll and the point found at the next successful poll is less than **X tolerance**.
- The change in the objective function from one successful poll to the next successful poll is less than **Function tolerance**.

The **Bind tolerance** option, which is used to identify active constraints for constrained problems, is not used as a stopping criterion.

4

Using the Genetic Algorithm

Overview of the Genetic Algorithm Tool (p. 4-2)	Provides an overview of the Genetic Algorithm Tool.
Using the Genetic Algorithm from the Command Line (p. 4-21)	Describes how to use the genetic algorithm at the command line.
Genetic Algorithm Examples (p. 4-30)	Explains how to set options for the genetic algorithm.

Overview of the Genetic Algorithm Tool

The section provides an overview of the Genetic Algorithm Tool. This section covers the following topics:

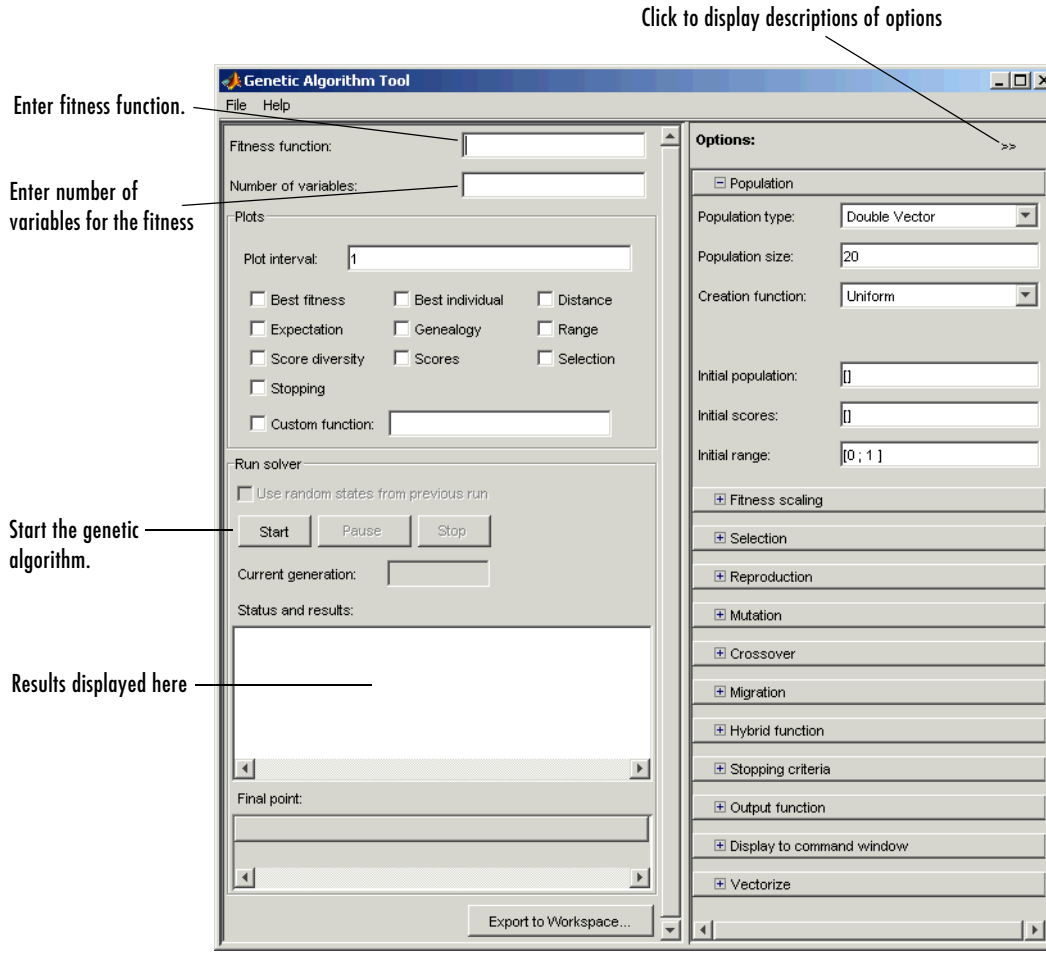
- "Opening the Genetic Algorithm Tool" on page 4-2
- "Defining a Problem in the Genetic Algorithm Tool" on page 4-3
- "Running the Genetic Algorithm" on page 4-4
- "Pausing and Stopping the Algorithm" on page 4-6
- "Displaying Plots" on page 4-7
- "Example — Creating a Custom Plot Function" on page 4-8
- "Reproducing Your Results" on page 4-11
- "Setting Options in the Genetic Algorithm Tool" on page 4-12
- "Importing and Exporting Options and Problems" on page 4-13
- "Example — Resuming the Genetic Algorithm from the Final Population" on page 4-16

Opening the Genetic Algorithm Tool

To open the tool, enter

```
gatool
```

at the MATLAB prompt. This opens the Genetic Algorithm Tool, as shown in the following figure.



Defining a Problem in the Genetic Algorithm Tool

You can define the problem you want to solve in the following two fields:

- **Fitness function** — The function you want to minimize. Enter a handle to an M-file function that computes the fitness function. “Writing M-Files for Functions You Want to Optimize” on page 1-3 describes how to write the M-file.

- **Number of variables** — The number of independent variables for the fitness function.

Note Do not use the Editor/Debugger to debug the M-file for the objective function while running the Genetic Algorithm Tool. Doing so results in Java exception messages in the Command Window and makes debugging more difficult. Instead, call the objective function directly from the command line or pass it to the genetic algorithm function `ga`. To facilitate debugging, you can export your problem from the Genetic Algorithm Tool to the MATLAB workspace, as described in "Importing and Exporting Options and Problems" on page 4-13.

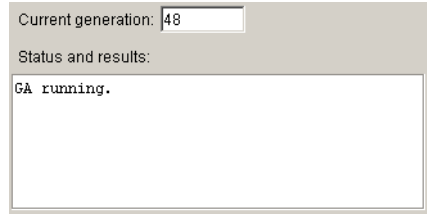
The following figure shows these fields for the example described in "Example: Rastrigin's Function" on page 2-6.

Running the Genetic Algorithm

To run the genetic algorithm, click **Start** in the **Run solver** pane. When you do so,

- The **Current generation** field displays the number of the current generation.
- The **Status and results** pane displays the message "`GA running.`".

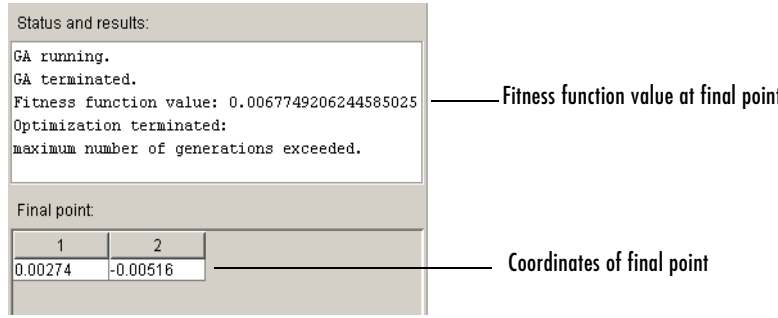
The following figure shows the **Current generation** field and **Status and results** pane while the algorithm is running.



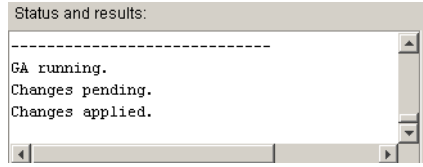
When the algorithm terminates, the **Status and results** pane displays

- The message "`GA terminated.`"
- The fitness function value of the best individual in the final generation
- The reason the algorithm terminated
- The coordinates of the final point

The following figure shows this information displayed when you run the example in "Example: Rastrigin's Function" on page 2-6.



You can change many of the settings in the Genetic Algorithm Tool while the algorithm is running. Your changes are applied at the next generation. Until your changes are applied, which occurs at the start of the next generation, the **Status and Results** pane displays the message `Changes pending`. At the start of the next generation, the pane displays the message `Changes applied`. as shown in the following figure.



Pausing and Stopping the Algorithm

While the genetic algorithm is running, you can

- Click **Pause** to temporarily suspend the algorithm. To resume the algorithm using the current population at the time you paused, click **Resume**.
- Click **Stop** to stop the algorithm. The **Status and results** pane displays the fitness function value of the best point in the current generation at the moment you clicked **Stop**.

Note If you click **Stop** and then run the genetic algorithm again by clicking **Start**, the algorithm begins with a new random initial population or with the population you specify in the **Initial population** field. If you want to restart the algorithm where it left off, use the **Pause** and **Resume** buttons.

"Example — Resuming the Genetic Algorithm from the Final Population" on page 4-16 explains what to do if you click **Stop** and later decide to resume the genetic algorithm from the final population of the last run.

Setting Stopping Criteria

The genetic algorithm uses five criteria, listed in the **Stopping criteria** options, to decide when to stop, in case you do not stop it manually by clicking **Stop**. The algorithm stops if any one of the following conditions occur:

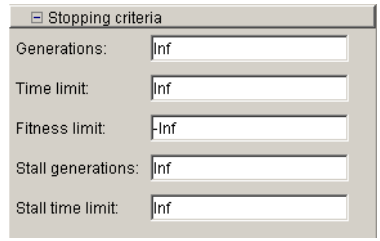
- **Generations** — The algorithm reaches the specified number of generations.
- **Time** — The algorithm runs for the specified amount of time in seconds.
- **Fitness limit** — The best fitness value in the current generation is less than or equal to the specified value.
- **Stall generations** — The algorithm computes the specified number of generations with no improvement in the fitness function.
- **Stall time limit** — The algorithm runs for the specified amount of time in seconds with no improvement in the fitness function.

If you want the genetic algorithm to continue running until you click **Pause** or **Stop**, you should change the default values of these options as follows:

- Set **Generations** to `Inf`

- Set **Time** to `Inf`.
- Set **Fitness limit** to `-Inf`.
- Set **Stall** generations to `Inf`.
- Set **Stall time limit** to `Inf`.

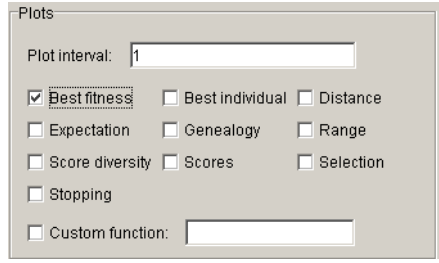
The following figure shows these settings.



Note Do not use these settings when calling the genetic algorithm function `ga` at the command line, as the function will never terminate until you press **Ctrl + C**. Instead, set **Generations** or **Time** limit to a finite number.

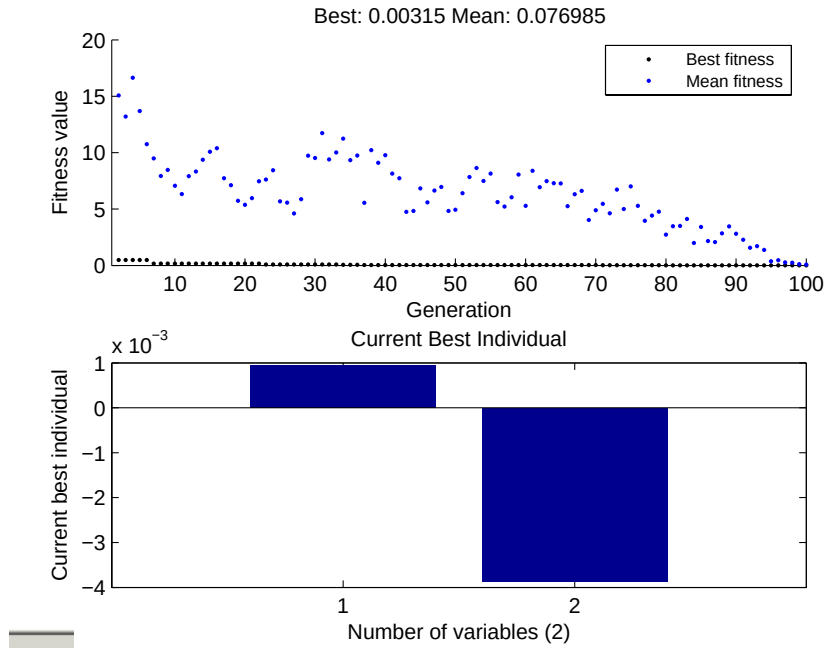
Displaying Plots

The **Plots** pane, shown in the following figure, enables you to display various plots of the results of the genetic algorithm.



Select the check boxes next to the plots you want to display. For example, if you select **Best fitness** and **Best individual**, and run the example described in

"Example: Rastrigin's Function" on page 2-6, the tool displays the plots shown in the following figure.



The upper plot displays the best and mean fitness values in each generation. The lower plot displays the coordinates of the point with the best fitness value in the current generation.

Note When you display more than one plot, clicking on any plot opens a larger version of it in a separate window.

"Plot Options" on page 6-4 describes the types of plots you can create.

Example — Creating a Custom Plot Function

If none of the plot functions that come with the toolbox is suitable for the output you want to plot, you can write your own custom plot function, which the

genetic algorithm calls at each generation to create the plot. This example shows how to create a plot function that displays the change in the best fitness value from the previous generation to the current generation.

This section covers the following topics:

- "Creating the Plot Function" on page 4-9
- "Using the Plot Function" on page 4-10
- "How the Plot Function Works" on page 4-10

Creating the Plot Function

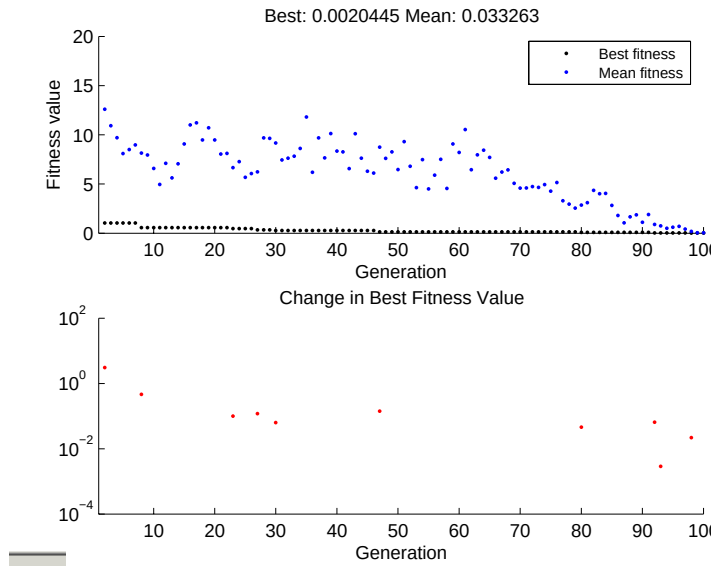
To create the plot function for this example, copy and paste the following code into a new M-file in the MATLAB Editor.

```
function state = gaplotchange(options, state, flag)
% GAPLOTCHANGE Plots the change in the best score from the
% previous generation.
%
persistent last_best % Best score in the previous generation

if(strcmp(flag,'init')) % Set up the plot
    set(gca,'xlim',[1,options.Generations],'Yscale','log');
    hold on;
    xlabel Generation
    title('Change in Best Fitness Value')
end

best = min(state.Score); % Best score in the current generation
if state.Generation == 0 % Set last_best to best.
    last_best = best;
else
   change = last_best - best; % Change in best score
   last_best=best;
   plot(state.Generation, change, '.r');
   title(['Change in Best Fitness Value'])
end
```

Then save the M-file as `gaplotchange.m` in a directory on the MATLAB path.

Using the Plot Function

To use the custom plot function, select **Custom** in the **Plots** pane and enter `@gaplotchange` in the field to the right. To compare the custom plot with the best fitness value plot, also select **Best fitness**. Now, if you run the example described in "Example: Rastrigin's Function" on page 2-6, the tool displays the plots shown in the following figure.



Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot only shows changes that are greater then 0. The logarithmic scale enables you to see small changes in the fitness function that the upper plot does not reveal.

How the Plot Function Works

The plot function uses information contained in the following structures, which the genetic algorithm passes to the function as input arguments:

- `options` — The current options settings
- `state` — Information about the current generation
- `flag` — String indicating the current status of the algorithm

The most important lines of the plot function are the following:

- `persistent last_best`

  Creates the persistent variable `last_best` — the best score in the previous generation. Persistent variables are preserved over multiple calls to the plot function.

- `set(gca,'xlim',[1,options.Generations],'Yscale','log');`

  Sets up the plot before the algorithm starts. `options.Generation` is the maximum number of generations.

- `best = min(state.Score)`

  The field `state.Score` contains the scores of all individuals in the current population. The variable `best` is the minimum score. For a complete description of the fields of the structure state, see "Structure of the Plot Functions" on page 6-5.

- `change = last_best - best`

  The variable change is the best score at the previous generation minus the best score in the current generation.

- `plot(state.Generation, change, '.r')`

  Plots the change at the current generation, whose number is contained in `state.Generation`.

The code for `gaplotchange` contains many of the same elements as the code for `gaplotbestf`, the function that creates the best fitness plot.

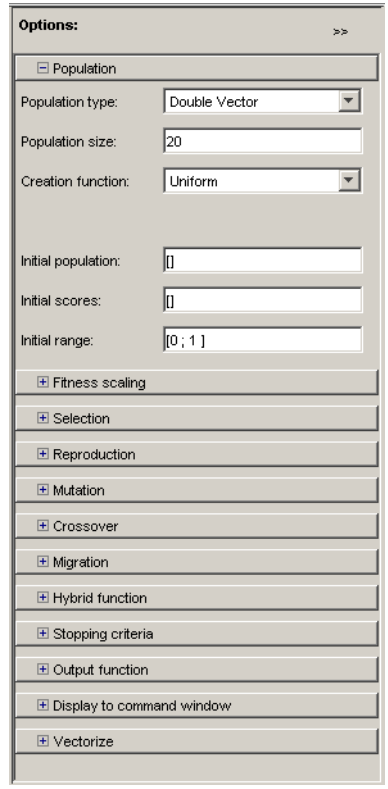
Reproducing Your Results

To reproduce the results of the last run of the genetic algorithm, select the **Use random states from previous run** check box. This resets the states of the random number generators used by the algorithm to their previous values. If you do not change any other settings in the Genetic Algorithm Tool, the next time you run the genetic algorithm, it returns the same results as the previous run.

Normally, you should leave **Use random states from previous run** unselected to get the benefit of randomness in the genetic algorithm. Select the **Use random states from previous run** check box if you want to analyze the results of that particular run or show the exact results to others.

Setting Options in the Genetic Algorithm Tool

You can set options for the genetic algorithm in the **Options** pane, shown in the figure below.



"Genetic Algorithm Examples" on page 4-30 describes how options settings affect the performance of the genetic algorithm. For a detailed description of all the available options, see "Genetic Algorithm Options" on page 6-3.

Setting Options as Variables in the MATLAB Workspace

You can set numerical options either directly, by typing their values in the corresponding edit box, or by entering the name of a variable in the MATLAB workspace that contains the option values. For example, you can set the **Initial point** to `[2.1 1.7]` in either of the following ways:

- Enter `[2.1 1.7]` in the **Initial point** field.
- Enter

  ```
  x0 = [2.1 1.7]
  ```

  at the MATLAB prompt and then enter `x0` in the **Initial point** field.

For options whose values are large matrices or vectors, it is often more convenient to define their values as variables in the MATLAB workspace. This way, it is easy to change the entries of the matrix or vector if necessary.

Importing and Exporting Options and Problems

You can export options and problem structures from the Genetic Algorithm Tool to the MATLAB workspace, and then import them back into the tool at a later time. This enables you to save the current settings for a problem and restore them later. You can also export the options structure and use it with the genetic algorithm function `ga` at the command line.

You can import and export the following information:

- The problem definition, including **Fitness function** and **Number of variables**
- The currently specified options
- The results of the algorithm

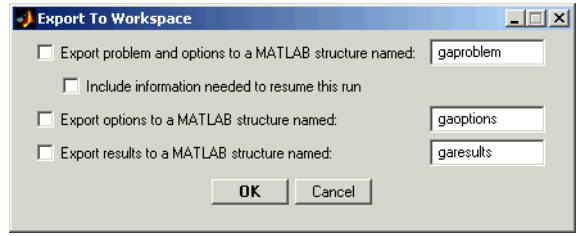
The following sections explain how to import and export this information:

- "Exporting Options and Problems" on page 4-13
- "Example — Running ga on an Exported Problem" on page 4-15
- "Importing Options" on page 4-16
- "Importing Problems" on page 4-16

Exporting Options and Problems

You can export options and problems to the MATLAB workspace so that you can use them at a future time in the Genetic Algorithm Tool. You can also apply the function `ga` using these options or problems at the command line — see "Using Options and Problems from the Genetic Algorithm Tool" on page 4-24.

To export options or problems, click the **Export** button or select **Export to Workspace** from the **File** menu. This opens the dialog box shown in the following figure.



The dialog provides the following options:

- To save both the problem definition and the current options settings, select **Export problem and options to a MATLAB structure named** and enter a name for the structure. Clicking **OK** saves this information to a structure in the MATLAB workspace. If you later import this structure into the Genetic Algorithm Tool, the settings for **Fitness function**, **Number of variables**, and all options settings are restored to the values they had when you exported the structure.

Note If you select **Use random states from previous run** in the **Run solver** pane before exporting a problem, the Genetic Algorithm Tool also saves the states of `rand` and `randn` at the beginning of the last run when you export. Then, when you import the problem and run the genetic algorithm with **Use random states from previous run** selected, the results of the run just before you exported the problem are reproduced exactly.

- If you want the genetic algorithm to resume from the final population of the last run before you exported the problem, select **Include information needed to resume this run**. Then, when you import the problem structure and click **Start**, the algorithm resumes from the final population of the previous run.

  To restore the genetic algorithm's default behavior of generating a random initial population, delete the population in the **Initial population** field and replace it with empty brackets, `[]`.

Note If you select **Include information needed to resume this run**, then selecting **Use random states from previous run** has no effect on the initial population created when you import the problem and run the genetic algorithm on it. The latter option is only intended to reproduce results from the beginning of a new run, not from a resumed run.

- To save only the options, select **Export options to a MATLAB structure named** and enter a name for the options structure.
- To save the results of the last run of the algorithm, select **Export results to a MATLAB structure named** and enter a name for the results structure.

Example — Running ga on an Exported Problem

To export the problem described in "Example: Rastrigin's Function" on page 2-6 and run the genetic algorithm function `ga` on it at the command line, do the following steps:

1. Click **Export to Workspace**.
2. In the **Export to Workspace** dialog box, enter a name for the problem structure, such as `my_gaproblem`, in the **Export problems and options to a MATLAB structure named** field.
3. At the MATLAB prompt, call the function `ga` with `my_gaproblem` as the input argument:

   ```
   [x fval] = patternsearch(my_gaproblem)
   ```

   This returns

   ```
   x =

       0.0027   -0.0052


   fval =

       0.0068
   ```

See "Using the Genetic Algorithm from the Command Line" on page 4-21 for form information.

Importing Options

To import an options structure from the MATLAB workspace, select **Import Options** from the **File** menu. This opens a dialog box that displays a list of the genetic algorithm options structures in the MATLAB workspace. When you select an options structure and click **Import**, the options fields in the Genetic Algorithm Tool are updated to display the values of the imported options.

You can create an options structure in either of the following ways:

- Calling `gaoptimset` with `options` as the output
- By saving the current options from the **Export to Workspace** dialog box in the Genetic Algorithm Tool

Importing Problems

To import a problem that you previously exported from the Genetic Algorithm Tool, select **Import Problem** from the **File** menu. This opens the dialog box that displays a list of the genetic algorithm problem structures in the MATLAB workspace. When you select a problem structure and click **OK**, the following fields are updated in the Genetic Algorithm Tool:

- **Fitness function**
- **Number of variables**
- The options fields

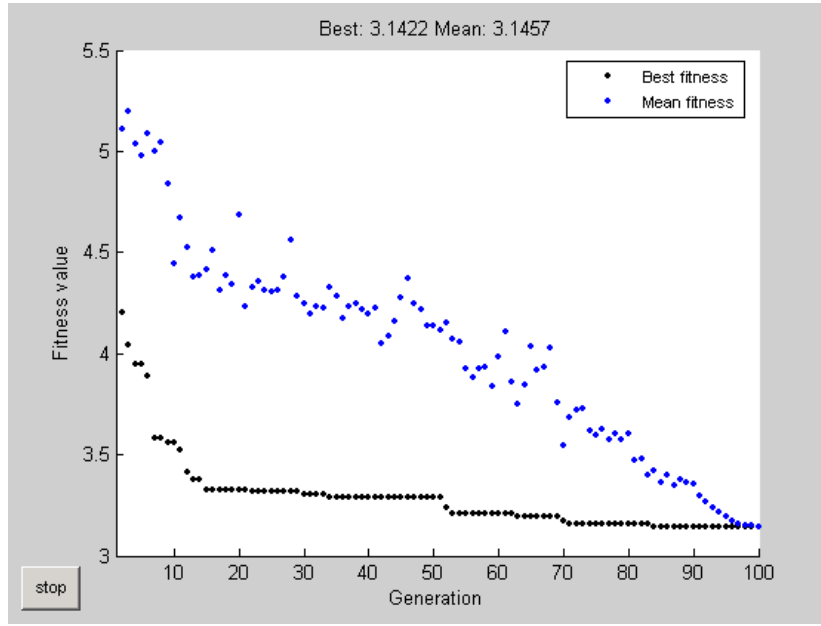
Example — Resuming the Genetic Algorithm from the Final Population

The following example shows how export a problem so that when you import it and click **Start**, the genetic algorithm resumes from the final population saved with the exported problem. To run the example, enter the following information in the Genetic Algorithm Tool:

- Set **Fitness function** to `@ackleyfcn`, which computes Ackley's function, a test function provided with the toolbox.
- Set **Number of variables** to `10`.

- Select **Best fitness** in the **Plots** pane.
- Click **Start**.

This displays the following plot.



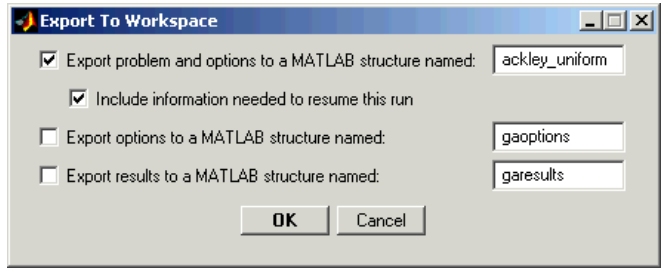
Suppose you want to experiment by running the genetic algorithm with other options settings, and then later restart this run from its final population with its current options settings. You can do this by the following steps:

1 Click the **Export to Workspace** button

2 In the dialog box that appears,

- Select **Export problem and options to a MATLAB structure named**.
- Enter a name for the problem and options, such as `ackley_run1`, in the text field.
- Select **Include information needed to resume this run**.

The dialog box should now appear as in the following figure.



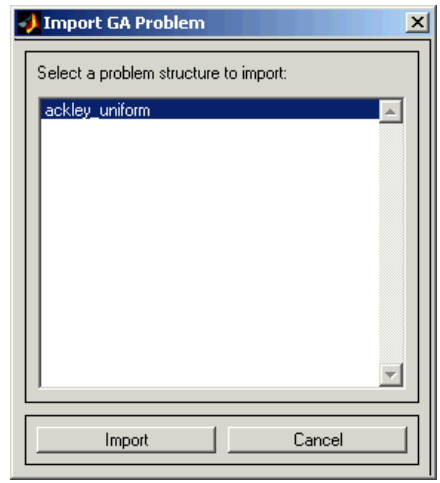
3 Click **OK**.

This exports the problem and options to a structure in the MATLAB workspace. You can view the structure in the MATLAB Command Window by entering

```
ackley_uniform

ackley_uniform =

      fitnessfcn: @ackleyfcn
    genomelength: 10
         options: [1x1 struct]
```

After running the genetic algorithm with different options settings or even a different fitness function, you can restore the problem as follows:

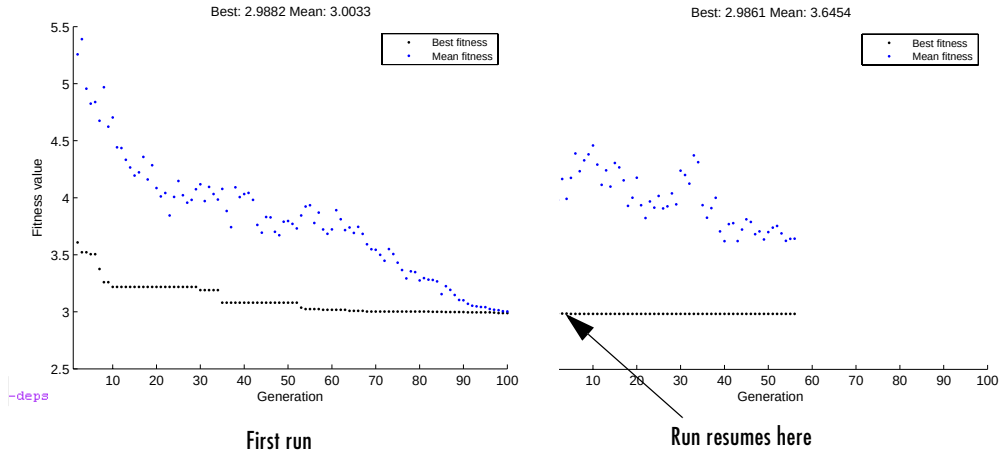
1 Select **Import Problem** from the **File** menu. This opens the dialog box shown in the following figure.



2 Select `ackley_uniform`.

3 Click **Import**.

This sets the **Initial population** field in **Population** options to the final population of the run before you exported the problem. All other options are restored to their setting during that run. When you click **Start**, the genetic algorithm resumes from the saved final population. The following figure shows the best fitness plots from the original run and the restarted run.



Note If, after running the genetic algorithm with the imported problem, you want to restore the genetic algorithm's default behavior of generating a random initial population, delete the population in the **Initial population** field and replace it with empty brackets, `[]`.

Generating an M-File

To create an M-file that runs the genetic algorithm, using the fitness function and options you specify in the Genetic Algorithm Tool, select **Generate M-File** from the **File** menu and save the M-file in a directory on the MATLAB path. Calling this M-file at the command line returns the same results as the Genetic Algorithm Tool, using the fitness function and options settings that were in place when you generated the M-file.

Using the Genetic Algorithm from the Command Line

As an alternative to using the Genetic Algorithm Tool, you can run the genetic algorithm function `ga` from the command line. This section explains how to do so and covers the following topics.

- "Running ga with the Default Options" on page 4-21
- "Setting Options for ga at the Command Line" on page 4-22
- "Using Options and Problems from the Genetic Algorithm Tool" on page 4-24
- "Reproducing Your Results" on page 4-25
- "Resuming ga from the Final Population of a Previous Run" on page 4-26
- "Running ga from an M-File" on page 4-27

Running ga with the Default Options

To run the genetic algorithm with the default options, call `ga` with the syntax

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

The input arguments to `ga` are

- `@fitnessfun` — A function handle to the M-file that computes the fitness function. "Writing M-Files for Functions You Want to Optimize" on page 1-3 explains how to write this M-file.
- `nvars` — The number of independent variables for the fitness function.

The output arguments are

- `x` — The final point
- `fval` — The value of the fitness function at `x`

For a description of additional output arguments, see the reference page for `ga`.

As an example, you can run the example described in "Example: Rastrigin's Function" on page 2-6 from the command line by entering

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

This returns

```
x =

    0.0027   -0.0052


fval =

    0.0068
```

Additional Output Arguments

To get more information about the performance of the genetic algorithm, you can call ga with the syntax

```
[x fval reason output population scores] = ga(@fitnessfcn, nvars)
```

Besides x and fval, this returns the following additional output arguments:

- `reason` — Reason the algorithm terminated
- `output` — Structure containing information about the performance of the algorithm at each generation
- `population` — Final population
- `scores` — Final scores

See the reference page for `ga` for more information about these arguments.

Setting Options for ga at the Command Line

You can specify any of the options that are available in the Genetic Algorithm Tool by passing an options structure as an input argument to `ga` using the syntax

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

You create the options structure using the function `gaoptimset`.

```
options = gaoptimset
```

This returns the structure `options` with the default values for its fields.

```
options =

            PopulationType: 'doubleVector'
          PopInitRange: [2x1 double]
        PopulationSize: 20
            EliteCount: 2
     CrossoverFraction: 0.8000
    MigrationDirection: 'forward'
     MigrationInterval: 20
     MigrationFraction: 0.2000
           Generations: 100
             TimeLimit: Inf
          FitnessLimit: -Inf
           StallLimitG: 50
           StallLimitS: 20
     InitialPopulation: []
         InitialScores: []
          PlotInterval: 1
           CreationFcn: @gacreationuniform
     FitnessScalingFcn: @fitscalingrank
          SelectionFcn: @selectionstochunif
          CrossoverFcn: @crossoverscattered
           MutationFcn: @mutationgaussian
             HybridFcn: []
               Display: 'final'
              PlotFcns: []
            OutputFcns: []
            Vectorized: 'off'
```

The function `ga` uses these default values if you do not pass in options as an input argument.

The value of each option is stored in a field of the options structure, such as `options.PopulationSize`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the size of the population for the genetic algorithm, enter

```
options.PopulationSize

ans =

    20
```

To create an options structure with a field value that is different from the default —for example to set `PopulationSize` to `100` instead of its default value `20` — enter

```
options = gaoptimset('PopulationSize', 100)
```

This creates the options structure with all values set to their defaults except for `PopulationSize`, which is set to `100`.

If you now enter,

```
ga(@fitnessfun, nvars, options)
```

`ga` runs the genetic algorithm with a population size of `100`.

If you subsequently decide to change another field in the options structure, such as setting `PlotFcns` to `@gaplotbestf`, which plots the best fitness function value at each generation, call `gaoptimset` with the syntax

```
options = gaoptimset(options, 'PlotFcns', @plotbestf)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@plotbestf`. Note that if you omit the input argument `options`, `gaoptimset` resets `PopulationSize` to its default value `20`.

You can also set both `PopulationSize` and `PlotFcns` with the single command

```
options = gaoptimset('PopulationSize',100,'PlotFcns',@plotbestf)
```

Using Options and Problems from the Genetic Algorithm Tool

As an alternative to creating an options structure using `gaoptimset`, you can set the values of options in the Genetic Algorithm Tool and then export the options to a structure in the MATLAB workspace, as described in “Exporting Options and Problems” on page 4-13. If you export the default options in the Genetic Algorithm Tool, the resulting structure `options` has the same settings as the default structure returned by the command

```
options = gaoptimset
```

If you export a problem structure, `ga_problem`, from the Genetic Algorithm Tool, you can apply `ga` to it using the syntax

```
[x fval] = ga(ga_problem)
```

The problem structure contains the following fields:

- `fitnessfcn` — Fitness function
- `nvars` — Number of variables for the problem
- `options` — Options structure

Reproducing Your Results

Because the genetic algorithm is stochastic — that is, it makes random choices — you get slightly different results each time you run the genetic algorithm. The algorithm uses the MATLAB uniform and normal random number generators, `rand` and `randn`, to makes random choices at each iteration. Each time `ga` calls `rand` and `randn`, their states are changed, so that the next time they are called, they return different random numbers. This is why the output of `ga` differs each time you run it.

If you need to reproduce your results exactly, you can call ga with an output argument that contains the current states of `rand` and `randn` and then reset the states to these values before running `ga` again. For example, to reproduce the output of `ga` applied to Rastrigin's function, call `ga` with the syntax

```
[x fval reason output] = ga(@rastriginsfcn, 2);
```

Suppose the results are

```
x =

    0.0027   -0.0052
fval =

    0.0068
```

The states of rand and randn are stored in the first two fields of output.

```
output =

       randstate: [35x1 double]
      randnstate: [2x1 double]
     generations: 100
       funccount: 2000
         message: [1x64 char]
```

Then, reset the states, by entering

```
rand('state', output.randstate);
randn('state', output.randnstate);
```

If you now run ga a second time, you get the same results.

Note If you do not need to reproduce your results, it is better not to set the states of rand and randn, so that you get the benefit of the randomness in the genetic algorithm.

Resuming ga from the Final Population of a Previous Run

By default, ga creates a new initial population each time you run it. However, you might get better results by using the final population from a previous run as the initial population for a new run. To do so, you must have saved the final population from the previous run by calling ga with the syntax

```
[x, fval, reason, output, final_pop] = ga(@fitnessfcn, nvars);
```

The last output argument, is the final population. To run ga using final_pop as the initial population, enter

```
options = gaoptimset('InitialPop', final_pop);
[x, fval, reason, output, final_pop2] = ga(@fitnessfcn, nvars);
```

If you want, you can then use final_pop2, the final population from the second run, as the initial population for a third run.

Running ga from an M-File

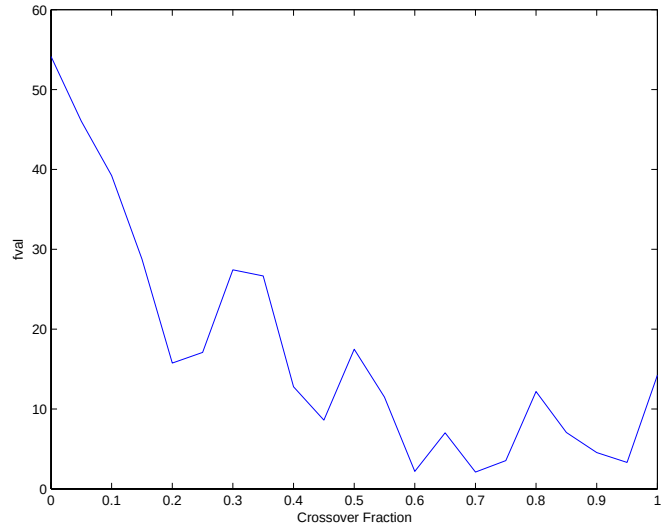
The command-line interface enables you to run the genetic algorithm many times, with different options settings, using an M-file. For example, you can run the genetic algorithm with different settings for **Crossover fraction** to see which one gives the best results. The following code runs the function `ga` twenty-one times, varying `options.CrossoverFraction` from `0` to `1` in increments of `0.5`, and records the results.

```
options = gaoptimset('Generations',300);
rand('state', 71); % These two commands are only included to
randn('state', 59); % make the results reproducible.
record=[];
for n=0:.05:1
   options = gaoptimset(options,'CrossoverFraction', n);
   [x fval]=ga(@rastriginsfcn, 10, options);
   record = [record; fval];
end
```

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

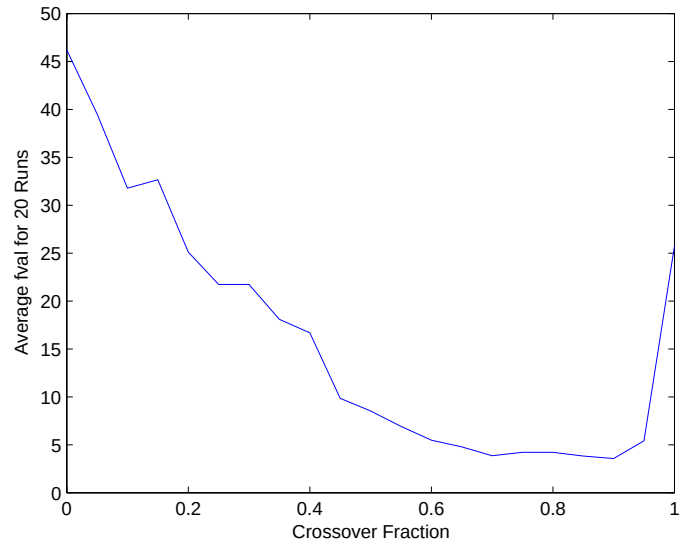
```
plot(0:.05:1, record);
xlabel('Crossover Fraction');
ylabel('fval')
```

This displays the following plot.



The plot indicates that you get the best results by setting `options.CrossoverFraction` to a value somewhere between `0.6` and `0.95`.

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. The following figure shows the resulting plot.



The plot narrows the range of best choices for `options.CrossoverFraction` to values between `0.7` and `0.9`.

Genetic Algorithm Examples

To get the best results from the genetic algorithm, you usually need to experiment with different options. Selecting the best options for a problem involves trial and error. This section describes some ways you can change options to improve results. For a complete description of the available options, see “Genetic Algorithm Options” on page 6-3.

This section covers the following topics:

- “Population Diversity” on page 4-30
- “Fitness Scaling” on page 4-35
- “Selection” on page 4-39
- “Reproduction Options” on page 4-40
- “Mutation and Crossover” on page 4-40
- “Setting the Amount of Mutation” on page 4-41
- “Setting the Crossover Fraction” on page 4-43
- “Example — Global Versus Local Minima” on page 4-48
- “Using a Hybrid Function” on page 4-52
- “Setting the Maximum Number of Generations” on page 4-54
- “Vectorizing the Fitness Function” on page 4-56

Population Diversity

One of the most important factors that determines the performance of the genetic algorithm performs 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. Getting the right amount of diversity is a matter of trial and error. If the diversity is too high or too low, the genetic algorithm might not perform well.

This section explains how to control diversity by setting the **Initial range** of the population. “Setting the Amount of Mutation” on page 4-41 describes how the amount of mutation affects diversity.

This section also explains how to set the population size.

Example —Setting the Initial Range

By default, the Genetic Algorithm Tool creates a random initial population using the creation function. You can specify the range of the vectors in the initial population in the **Initial range** field in **Population** options.

Note The initial range only restricts the range of the points in the initial population. Subsequent generations can contain points whose entries do not lie in the initial range.

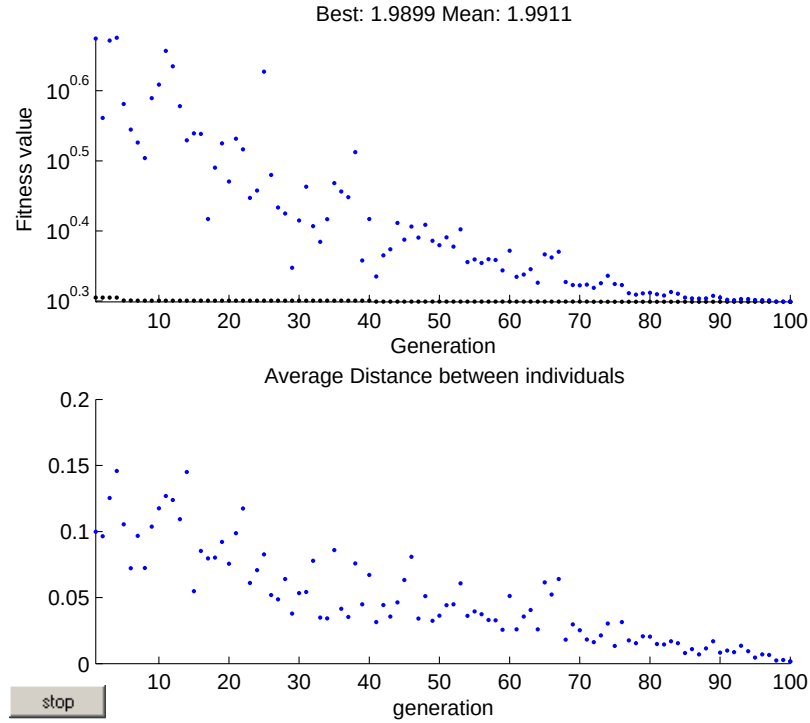
If you know approximately where the solution to a problem lies, you should specify the initial range so that it contains your guess for the solution. However, the genetic algorithm can find the solution even if it does not lie in the initial range, provided that the populations have enough diversity.

The following example shows how the initial range affects the performance of the genetic algorithm. The example uses Rastrigin's function, described in "Example: Rastrigin's Function" on page 2-6. The minimum value of the function is 0, which occurs at the origin.

To run the example, make the following settings in the Genetic Algorithm Tool:

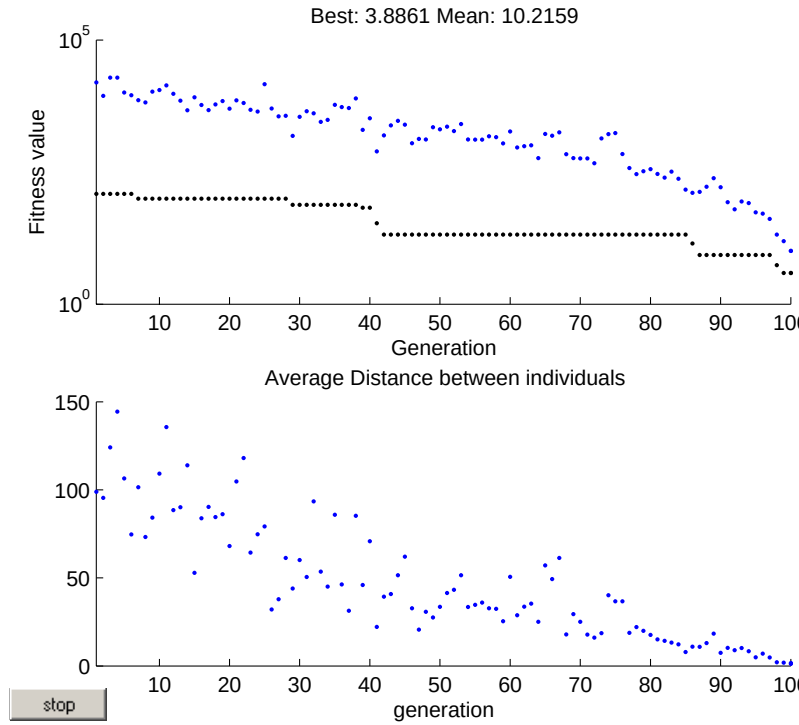
- Set **Fitness function** to `@Rastriginsfcn`.
- Set **Number of variables** to `2`.
- Select **Best fitness** in the **Plots** pane.
- Select **Distance** in the **Plots** pane.
- Set **Initial range** to `[1; 1.1]`.

Then click **Start**. The genetic algorithm returns the best fitness function value of approximately `2` and displays the plots in the following figure.



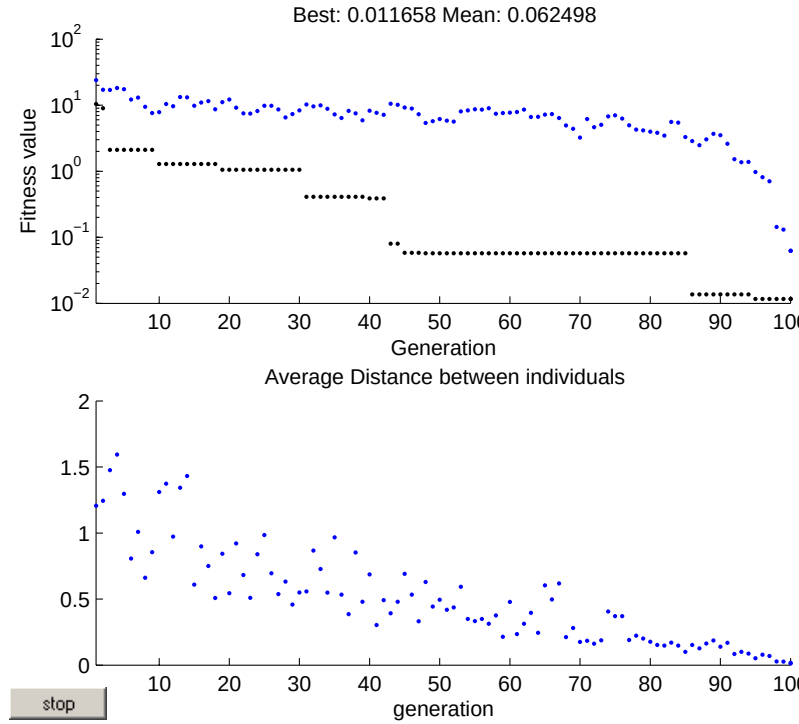
The upper plot the best fitness values at each generation shows little progress in lowering the fitness function. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too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

Next, try setting **Initial range** to `[1; 100]` and running the algorithm. The genetic algorithm returns the best fitness value of approximately 3.9 and displays the following plots.



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. This returns the best fitness value of approximately `.012` and displays the following plots.



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

Setting the Population Size

The **Size** field in **Population** options determines the size of the population at each generation. Increasing the population size enables the genetic algorithm to search more points and thereby obtain a better result. However, the larger the population size, the longer the genetic algorithm takes to compute each generation.

Note You should set **Size** to be at least the value of **Number of variables**, so that the individuals in each population span the space being searched.

You can experiment with different settings for **Population size** that return good results without taking a prohibitive amount of time to run.

Fitness Scaling

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation. The selection function assigns a higher probability of selection to individuals with higher scaled values.

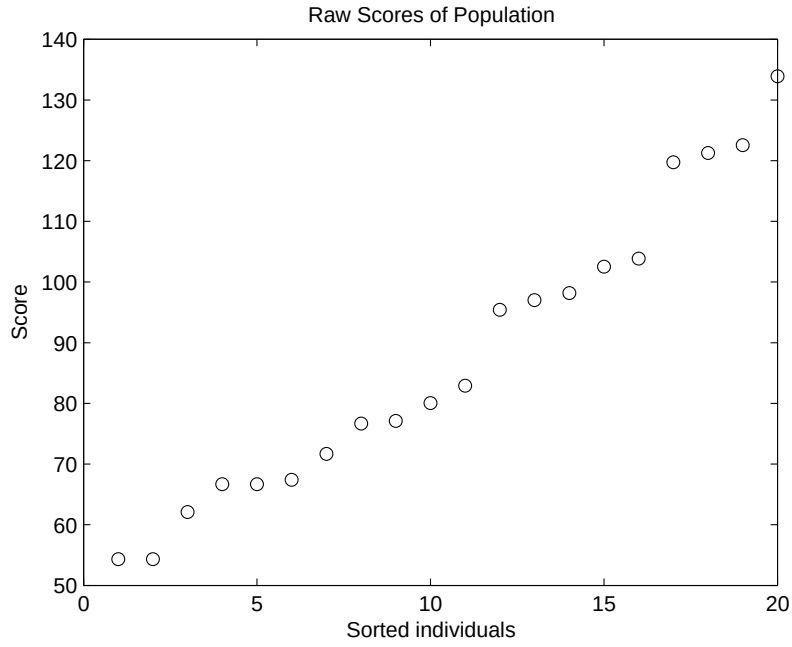
The range of the scaled values affects the performance of the genetic algorithm. If the scaled values vary too widely, the individuals with the highest scaled values reproduce too rapidly, taking over the population gene pool too quickly, and preventing the genetic algorithm from searching other areas of the solution space. On the other hand, if the scaled values vary only a little, all individuals have approximately the same chance of reproduction and the search will progress very slowly.

The default fitness scaling function, `Rank`, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores: the rank of the most fit individual is 1, the next most fit is 2, and so on. The rank scaling function assigns scaled values so that

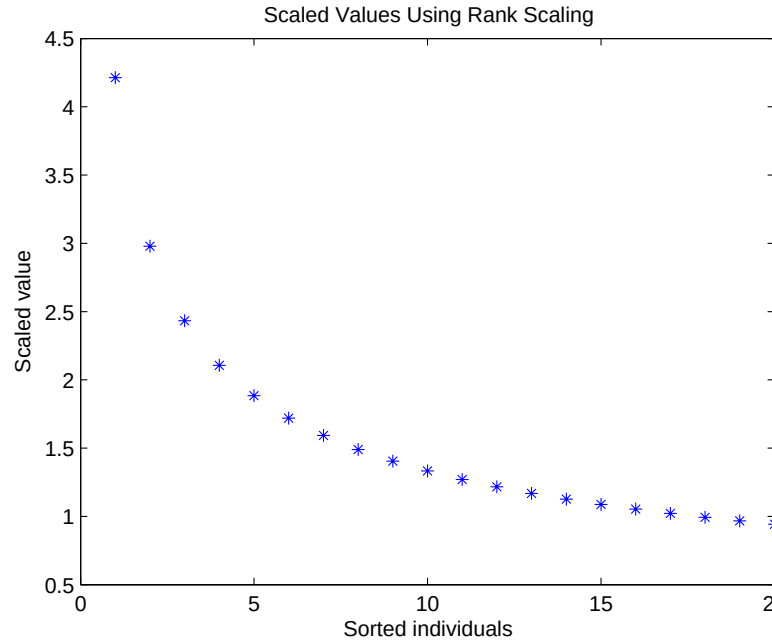
- The scaled value of an individual with rank n is proportional to $1/(\sqrt{n})$.
- The sum of the scaled values over the entire population equals the number of parents needed to create the next generation.

Rank fitness scaling removes the effect of the spread of the raw scores.

The following plot shows the raw scores of a typical population of 20 individuals, sorted in increasing order.


Raw Scores of Population
Score
Sorted individuals
140
130
120
110
100
90
80
70
60
50
0
5
10
15
20

The following plot shows the scaled values of the raw scores using rank scaling.

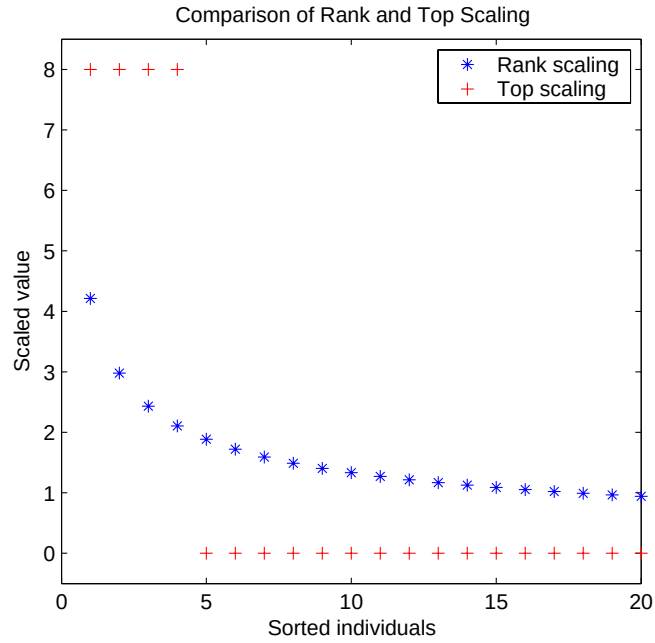


Because the algorithm minimizes the fitness function, lower raw scores have higher scaled values. Also, because rank scaling assigns values that depend only on an individual's rank, the scaled values shown would be the same for any population of size 20 and number of parents equal to 32.

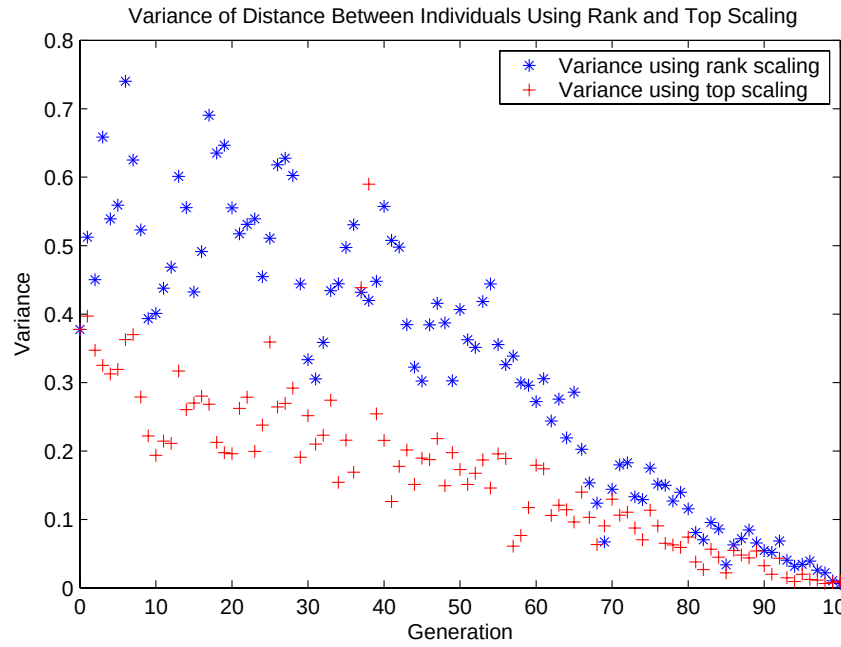
Comparing Rank and Top Scaling

To see the effect of scaling, you can compare the results of the genetic algorithm using rank scaling with one of the other scaling functions, such as `Top`. By default, top scaling assigns the four fittest individuals the same scaled value, equal to the number of parents divided by 4, and assigns the rest the value 0. Using the default selection function, only the four fittest individuals can be selected as parents.

The following figure compares the scaled values of a population of size 20 with number of parents equal to 32 using rank and top scaling.



Because top scaling restricts parents to the fittest individuals, it creates less diverse populations than rank scaling. The following plot compares the variances of distances between individuals at each generation using rank and top scaling.



Selection

The selection function chooses parents for the next generation based on their scaled values from the fitness scaling function. An individual can be selected more than once as a parent, in which case it contributes its genes to more than one child. The default selection function, `Stochastic uniform`, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on.

A more deterministic selection function is `Remainder`, which performs two steps:

- In the first step, the function selects parents deterministically according to the integer part of the scaled value for each individual. For example, if an individual's scaled value is 2.3, the function selects that individual twice as a parent.

- In the second step, the selection function selects additional parents using the fractional parts of the scaled values, as in stochastic uniform selection. The function lays out a line in sections, whose lengths are proportional to the fractional part of the scaled value of the individuals, and moves along the line in equal steps to select the parents.

  Note that if the fractional parts of the scaled values all equal 0, as can occur using `Top` scaling, the selection is entirely deterministic.

Reproduction Options

Reproduction options control how the genetic algorithm creates the next generation. The options are

- **Elite count** — The number of individuals with the best fitness values in the current generation that are guaranteed to survive to the next generation. These individuals are called *elite children*. The default value of **Elite count** is 2.

  When **Elite count** is at least 1, the best fitness value can only decrease from one generation to the next. This is what you want to happen, since the genetic algorithm minimizes the fitness function. Setting **Elite count** to a high value causes the fittest individuals to dominate the population, which can make the search less effective.
- **Crossover fraction** — The fraction of individuals in the next generation, other than elite children, that are created by crossover."Setting the Crossover Fraction" on page 4-43 describes how the value of **Crossover fraction** affects the performance of the genetic algorithm.

Mutation and Crossover

The genetic algorithm uses the individuals in the current generation to create the children that make up the next generation. Besides elite children, which correspond to the individuals in the current generation with the best fitness values, the algorithm creates

- Crossover children by selecting vector entries, or genes, from a pair of individuals in the current generation and combines them to form a child.
- Mutation children by applying random changes to a single individual in the current generation to create a child.

Both processes are essential to the genetic algorithm. Crossover enables the algorithm to extract the best genes from different individuals and recombine them into potentially superior children. Mutation adds to the diversity of a population and thereby increases the likelihood that the algorithm will generate individuals with better fitness values. Without mutation, the algorithm could only produce individuals whose genes were a subset of the combined genes in the initial population.

See "Creating the Next Generation" on page 2-20 for an example of how the genetic algorithm applies mutation and crossover.

You can specify how many of each type of children the algorithm creates as follows:

- **Elite count**, in **Reproduction** options, specifies the number of elite children.
- **Crossover fraction**, in **Reproduction** options, specifies the fraction of the population, other than elite children, that are crossover children.

For example, if the **Population size** is `20`, the **Elite count** is `2`, and the **Crossover fraction** is `0.8`, the numbers of each type of children in the next generation is as follows:

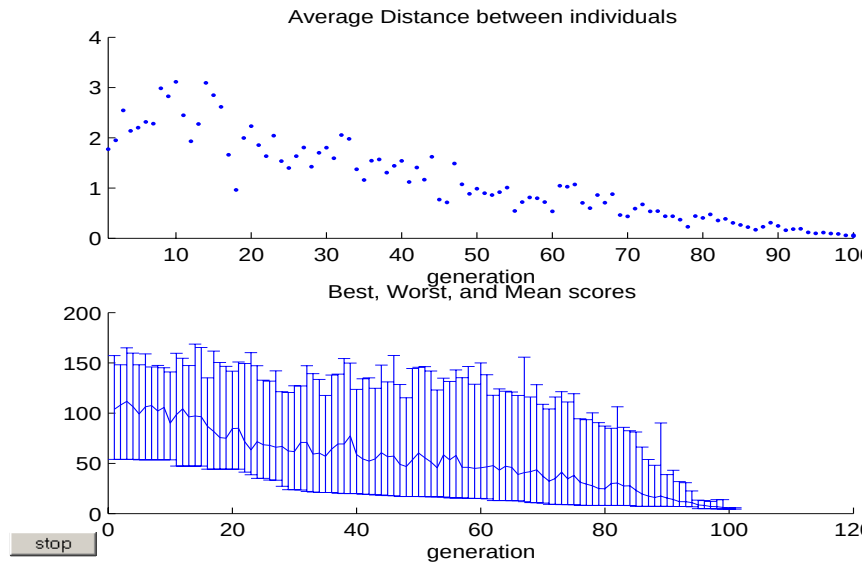
- There are 2 elite children
- There are 18 individuals other than elite children, so the algorithm rounds 0.8*18 = 14.4 to 14 to get the number of crossover children.
- The remaining 4 individuals, other than elite children, are mutation children.

Setting the Amount of Mutation

The genetic algorithm applies mutations using the function that you specify in the **Mutation function** field. The default mutation function, `Gaussian`, adds a random number, or *mutation*, chosen from a Gaussian distribution, to each entry of the parent vector. Typically, the amount of mutation, which is proportional to the standard deviation of the distribution, decreases as at each new generation. You can control the average amount of mutation that the algorithm applies to a parent in each generation through the **Scale** and **Shrink** options:

- **Scale** controls the standard deviation of the mutation at the first generation, which is **Scale** multiplied by the range of the initial population, which you specify by the **Initial range** option.
- **Shrink** controls the rate at which the average amount of mutation decreases. The standard deviation decreases linearly so that its final value equals 1 - **Shrink** times its initial value at the first generation. For example, if **Shrink** has the default value of 1, then the amount of mutation decreases to 0 at the final step.

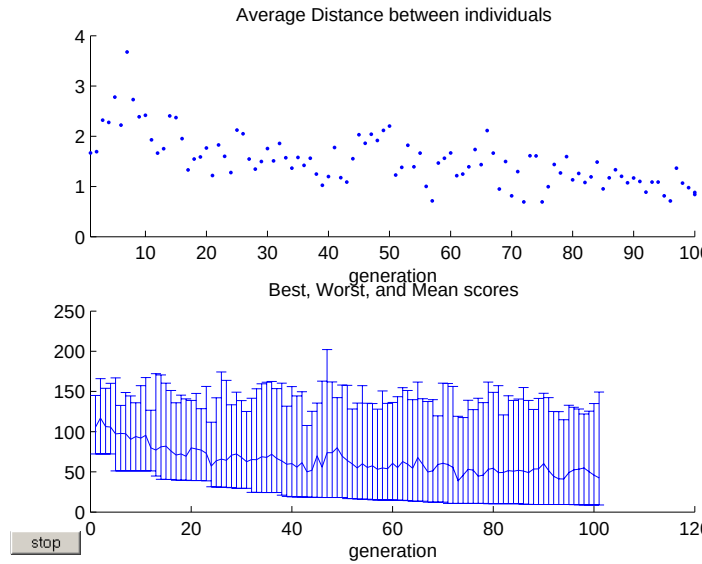
You can see the effect of mutation by selecting the plot functions **Distance** and **Range**, and then running the genetic algorithm on a problem such as the one described in "Example: Rastrigin's Function" on page 2-6. The following figure shows the plot.



The upper plot displays the average distance between points in each generation. As the amount of mutation decreases, so does the average distance between individuals, which is approximately 0 at the final generation. The lower plot displays a vertical line at each generation, showing the range from the smallest to the largest fitness value, as well as mean fitness value. As the amount of mutation decreases, so does the range. These plots show that

reducing the amount of mutation decreases the diversity of subsequent generations.

For comparison, the following figure shows the plots for **Distance** and **Range** when you set **Shrink** to `0.5`.



With **Shrink** set to `0.5`, the average amount of mutation decreases by a factor of 1/2 by the final generation. As a result, the average distance between individuals decreases by approximately the same factor.

Setting the Crossover Fraction

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.

The example uses the fitness function whose value at a point is the sum of the absolute values of the coordinates at the points. That is,

$$f(x_1, x_2, \ldots, x_n) = |x_1| + |x_2| + \ldots + |x_n|$$

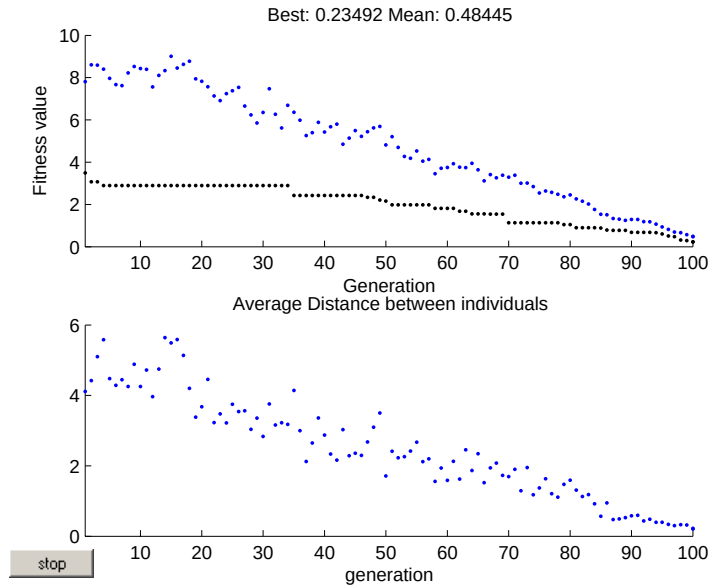
You can define this function as an anonymous function by setting **Fitness function** to

```
@(x) sum(abs(x))
```

To run the example,

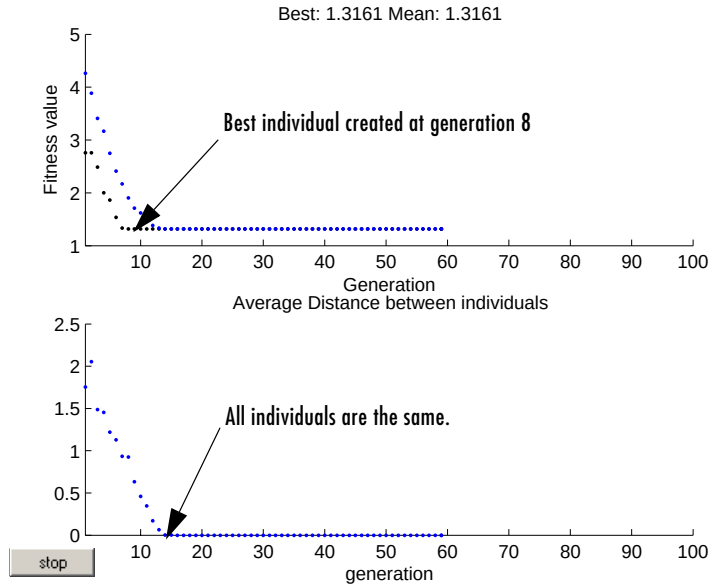
- Set **Fitness function** to `@(x) sum(abs(x))`.
- Set **Number of variables** to `10`.
- Set **Initial range** to `[-1; 1]`.
- Select the **Best fitness** and **Distance** in the **Plots** pane.

First, run the example with the default value of `0.8` for **Crossover fraction**. This returns the best fitness value of approximately `0.2` and displays the following plots.



Crossover Without Mutation

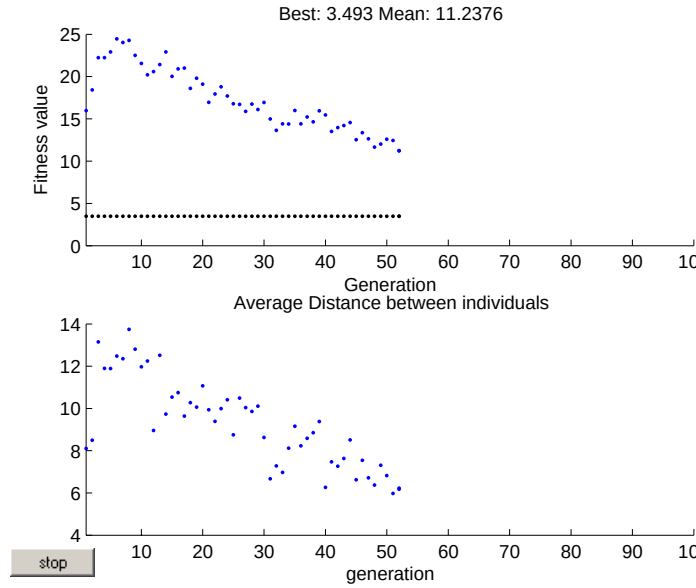
To see how the genetic algorithm performs when there is no mutation, set **Crossover fraction** to `1.0` and click **Start**. This returns the best fitness value of approximately `1.3` and displays the following plots.



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, where the best fitness plot becomes level. After this, it creates new copies of the best individual, which are then are 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 after generation 8, 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 of approximately `3.5` and displays the following plots.



In this case, the random changes that the algorithm applies never improve the fitness value of the best individual at the first generation. 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 is level and the algorithm stalls at generation number 50.

Comparing Results for Varying Crossover Fractions

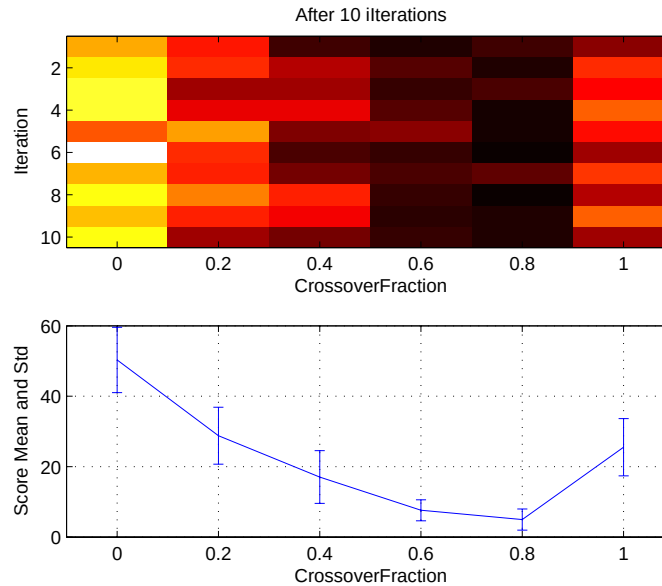
The demo `deterministicstudy.m`, which is included in the toolbox, compares the results of applying the genetic algorithm to Rastrigin's function with **Crossover fraction** set to `0`, `.2`, `.4`, `.6`, `.8`, and `1`. The demo runs for 10 generations. At each generation, the demo plots the means and standard

deviations of the best fitness values in all the preceding generations, for each value of the **Crossover fraction**.

To run the demo, enter

```
deterministicstudy
```

at the MATLAB prompt. When the demo 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.

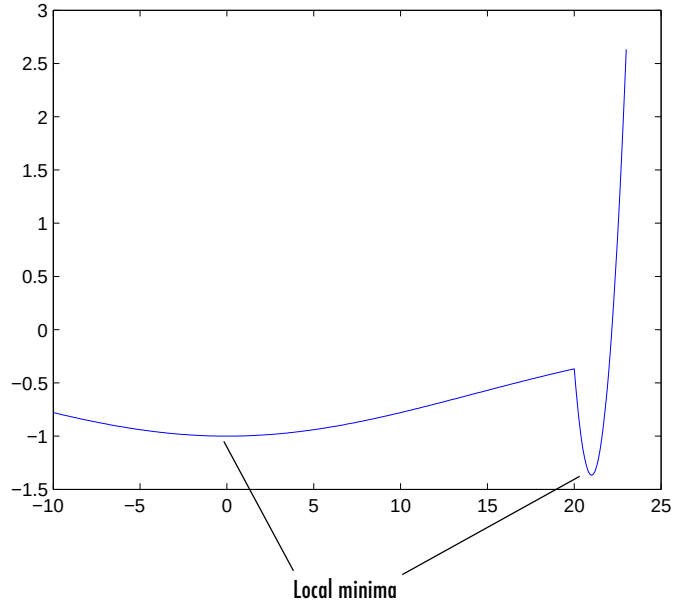
Example — Global Versus Local Minima

Sometimes the goal of an optimization is to find the global minimum or maximum of a function — a point where the function value is smaller or larger at any other point in the search space. However, optimization algorithms sometimes return a local minimum — a point where the function value is smaller than at nearby points, but possibly greater than at a distant point in the search space. The genetic algorithm can sometimes overcome this deficiency with the right settings.

As an example, consider the following function

$$f(x) = \begin{cases} -\exp\left(-\left(\frac{x}{20}\right)^2\right) & \text{for } x \le 20 \\ -\exp(-1) + (x-20)(x-22) & \text{for } x > 20 \end{cases}$$

The following figure shows a plot of the function.



The function has two local minima, one at x = 0, where the function value is - 1, and the other at x = 21, where the function value is - 1 - 1/e. Since the latter value is smaller, the global minimum occurs at x = 21.

Running the Genetic Algorithm on the Example

To run the genetic algorithm on this example,

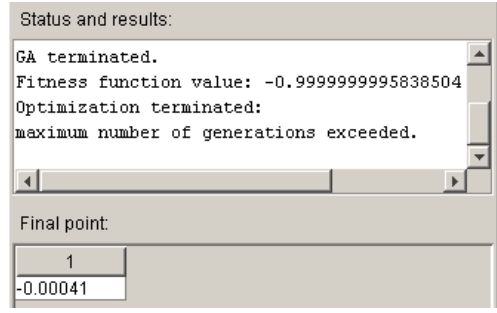
1. Copy and paste the following code into a new M-file in the MATLAB Editor.

```
function y = two_min(x)
if x<20
    y = -exp(-(x/20).^2);
else
    y = -exp(-1)+(x-20)*(x-22);
end
```

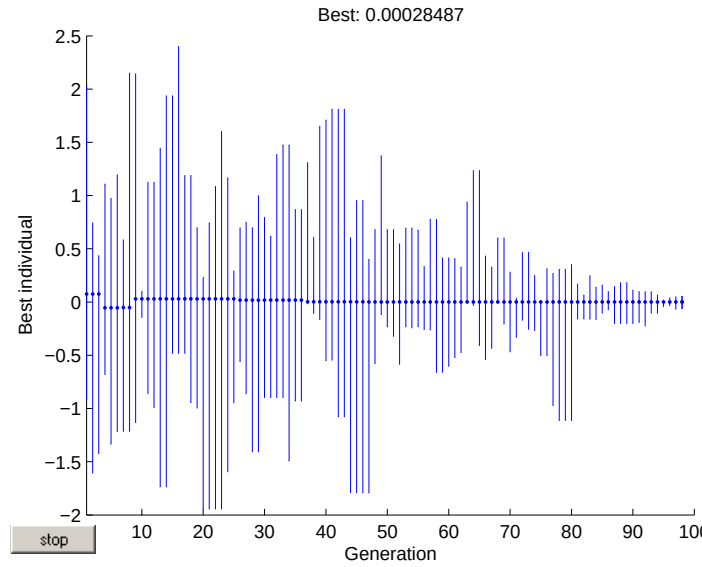
2. Save the file as `two_min.m` in a directory on the MATLAB path.

3. In the Genetic Algorithm Tool,
   - Set **Fitness function** to `@two_min`.
   - Set **Number of variables** to `1`.
   - Click **Start**.

The genetic algorithm returns a point very close to the local minimum at x = 0.

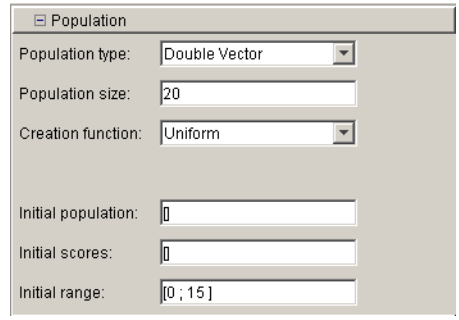


The following custom plot shows why the algorithm finds the local minimum rather than the global minimum. The plot shows the range of individuals in each generation and the best individual.



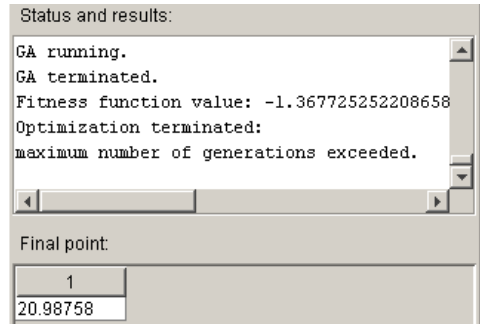
Note that all individuals are between -2 and 2.5. While this range is larger than the default **Initial range** of `[0;1]`, due to mutation, it is not large enough to explore points near the global minimum at x = 21.

One way to make the genetic algorithm explore a wider range of points — that is, to increase the diversity of the populations — is to increase the **Initial range**. The **Initial range** does not have to include the point x = 21, but it must be large enough so that the algorithm generates individuals near x = 21. Set **Initial range** to `[0;15]` as shown in the following figure.

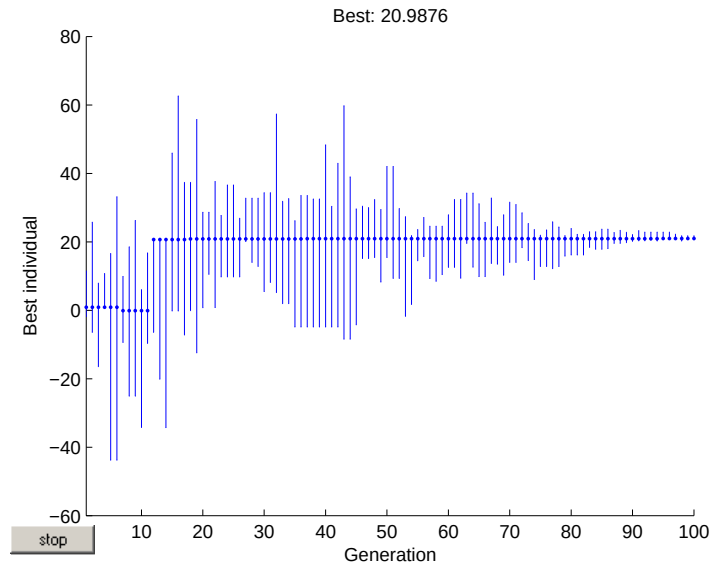


Set **Initial range** to `[0; 15]`.

Then click **Start**. The genetic algorithm returns a point very close 21.



This time, the custom plot shows a much wider range of individuals. By the second generation there are individuals greater than 21, and by generation 12, the algorithm finds a best individual that is approximately equal to 21.



Using a Hybrid Function

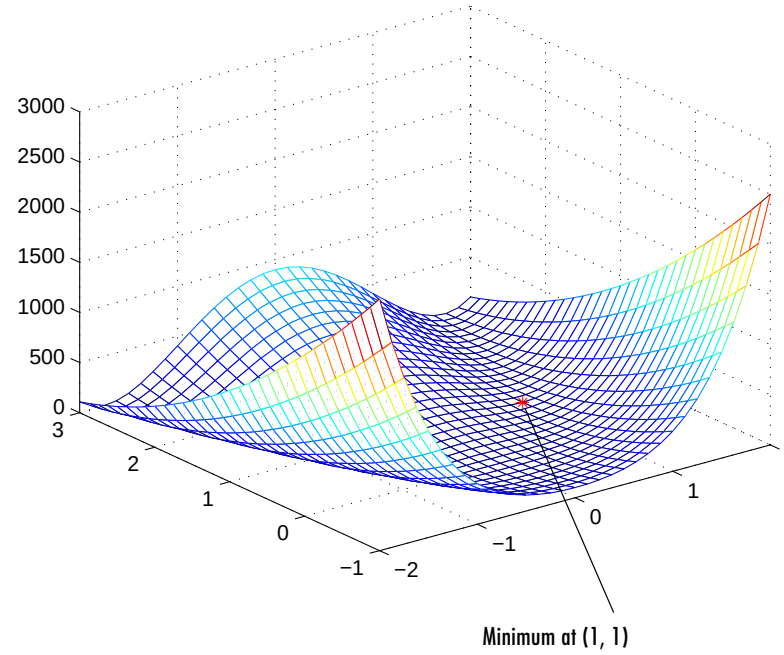
A hybrid function is an optimization function that runs after the genetic algorithm terminates in order to improve the value of the fitness function. The hybrid function uses the final point from the genetic algorithm as its initial point. You can specify a hybrid function in **Hybrid function** options.

This example uses the function `fminunc`, an unconstrained minimization function in the Optimization Toolbox. The example first runs the genetic algorithm to find a point close to the optimal point and then uses that point as the initial point for `fminunc`.

The example finds the minimum of Rosenbrock's function, which is defined by

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The following figure shows a plot of Rosenbrock's function.



The toolbox provides an M-file, `dejong2fcn.m`, that computes the function. To a see a demo of this example, enter

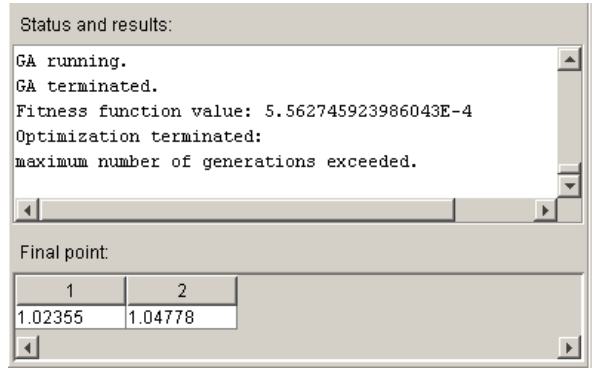
```
hybriddemo
```

at the MATLAB prompt.

To explore the example, first enter `gatool` to open the Genetic Algorithm Tool and enter the following settings:

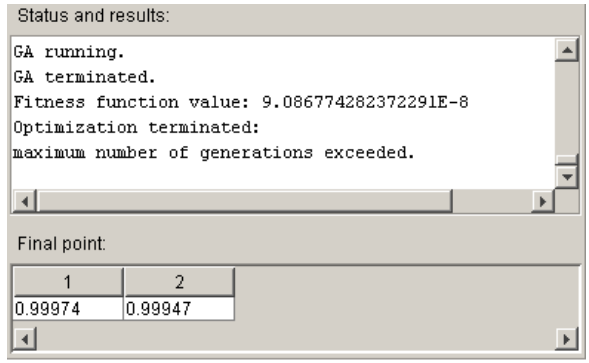
- Set **Fitness function** to `@dejong2fcn`.
- Set **Number of variables** to `2`.
- Set **Population size** to 10.

Before adding a hybrid function, trying running the genetic algorithm by itself, by clicking **Start**. The genetic algorithm displays the following results in the **Status and results** pane.



The final point is close to the true minimum at (1, 1). You can improve this result by setting **Hybrid function** to `fminunc` in **Hybrid function** options.

When the genetic algorithm terminates, the function `fminunc` takes the final point of the genetic algorithm and as its initial point and returns a more accurate result, as shown in the **Status and results** pane.



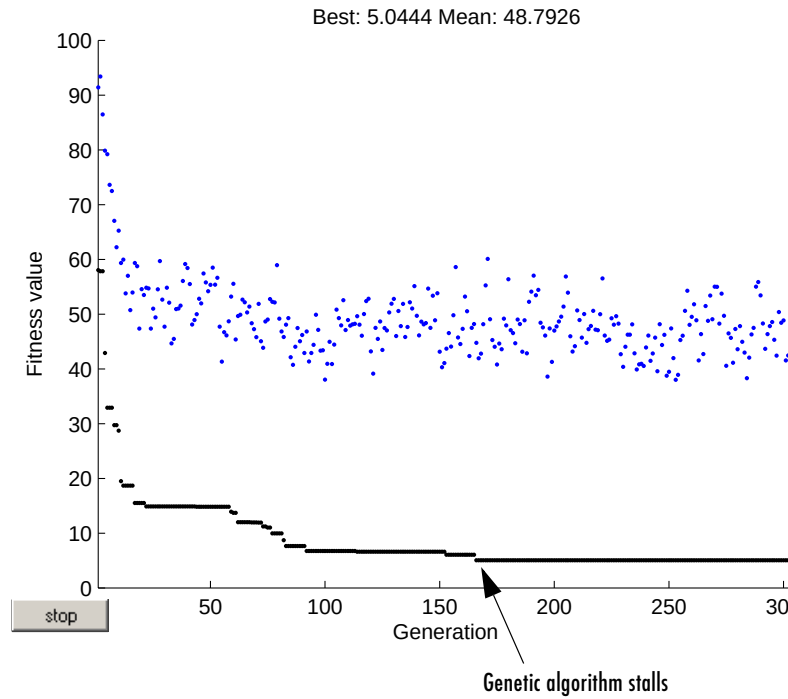
Setting the Maximum Number of Generations

The **Generations** option in **Stopping criteria** determines the maximum number of generations the genetic algorithm runs for — see “Stopping Conditions for the Algorithm” on page 2-23. Increasing **Generations** often improves the final result.

As an example, change the settings in the Genetic Algorithm Tool as follows:

- Set **Fitness function** to `@rastriginsfcn`.
- Set **Number of variables** to `10`.
- Select **Best fitness** in the **Plots** pane.
- Set **Generations** to `Inf`.
- Set **Stall generations** to `Inf`.
- Set **Stall time** to `Inf`

Then run the genetic algorithm for approximately 300 generations and click **Stop**. The following figure shows the resulting **Best fitness** plot after 300 generations.



Note that the algorithm *stalls* at approximately generation number 170 — that is, there is no immediate improvement in the fitness function after generation 170. If you restore **Stall generations** to its default value of 50, the algorithm would terminate at approximately generation number 230. If the genetic algorithm stalls repeated with the current setting for **Generations**, you can try increasing both **Generations** and **Stall generations** to improve your results. However, changing other options might be more effective.

Note When **Mutation function** is set to `Gaussian`, increasing the value of **Generations** might actually worsen the final result. This can occur because the Gaussian mutation function decreases the average amount of mutation in each generation by a factor that depends on **Generations**. Consequently, the setting for **Generations** affects the behavior of the algorithm.

Vectorizing the Fitness Function

The genetic algorithm usually runs faster if you *vectorize* the fitness function. This means that the genetic algorithm only calls the fitness function once, but expects the fitness function to compute the fitness for all individuals in the current population at once. To vectorize the fitness function,

- Write the M-file that computes the function so that it accepts a matrix with arbitrarily many rows, corresponding to the individuals in the population. For example, to vectorize the function

  $$f(x_1, x_2) = x_1^2 - 2x_1x_2 + 6x_1 + x_2^2 - 6x_2$$

  write the M-file using the following code:

  ```
  z =x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + x(:,2).^2 - 6*x(:,2);
  ```

  The colon in the first entry of x indicates all the rows of `x`, so that `x(:, 1)` is a vector. The `.^` and `.*` operators perform element-wise operations on the vectors.
- Set the **Vectorize** option to `On`.

Note The fitness function must accept an arbitrary number of rows to use the **Vectorize** option.

The following comparison, run at the command line, shows the improvement in speed with **Vectorize** set to `On`.

```
tic;ga(@rastriginsfcn,20);toc

elapsed_time =

    4.3660
options=gaoptimset('Vectorize','on');
tic;ga(@rastriginsfcn,20,options);toc

elapsed_time =

    0.5810
```

5

Using Direct Search

Overview of the Pattern Search Tool (p. 5-2)	Provides an overview of the Pattern Search Tool.
Performing a Pattern Search from the Command Line (p. 5-14)	Explains how to perform a pattern search from the command line.
Pattern Search Examples (p. 5-19)	Explains how to set options for a pattern search.
Parameterizing Functions Called by patternsearch or ga (p. 5-42)	Explains how to write functions with additional parameters

Overview of the Pattern Search Tool

The section provides an overview of the Pattern Search Tool, the graphical user interface (GUI) for performing a pattern search. This section covers the following topics:

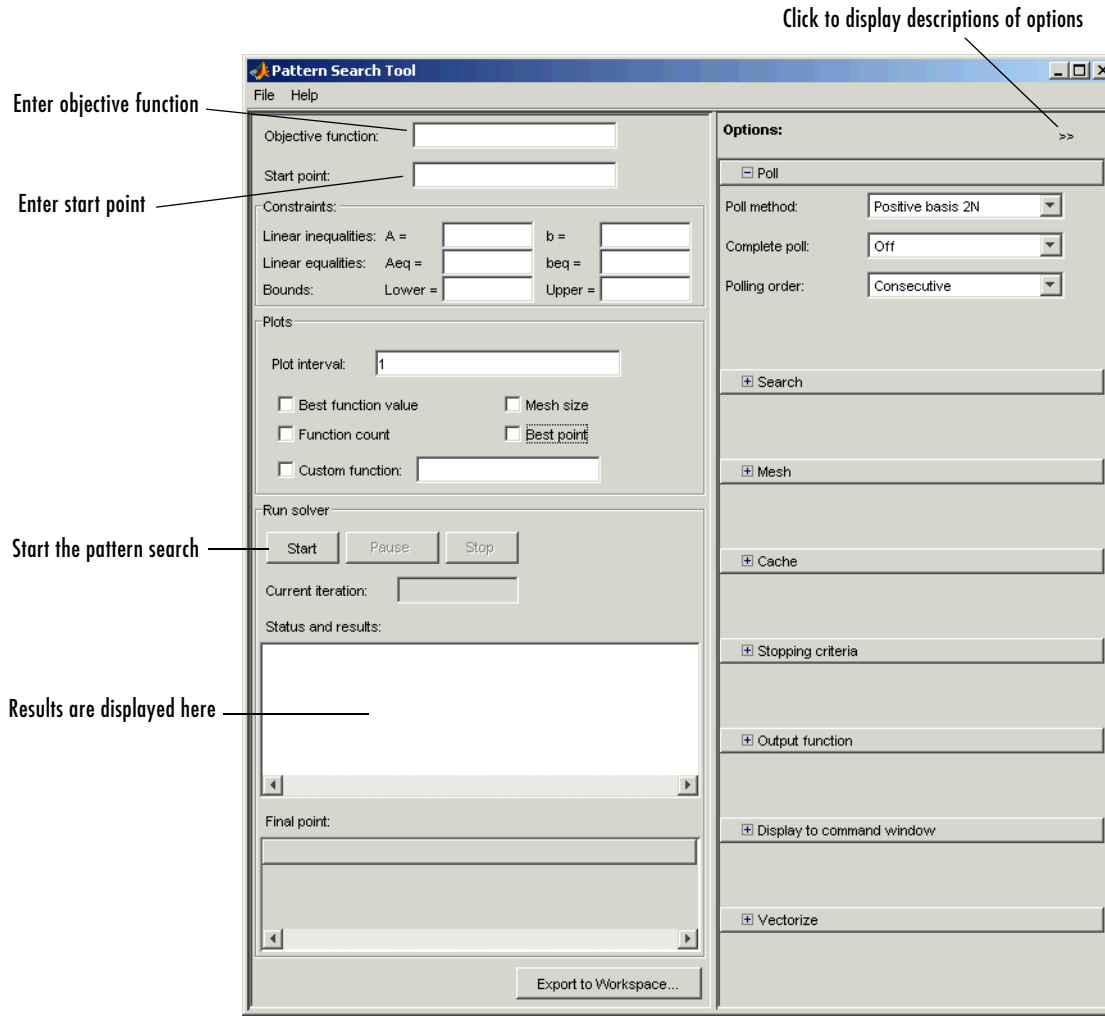
- "Opening the Pattern Search Tool" on page 5-2
- "Defining a Problem in the Pattern Search Tool" on page 5-3
- "Running a Pattern Search" on page 5-5
- "Example — A Constrained Problem" on page 5-6
- "Pausing and Stopping the Algorithm" on page 5-8
- "Displaying Plots" on page 5-8
- "Setting Options in the Pattern Search Tool" on page 5-10
- "Importing and Exporting Options and Problems" on page 5-11
- "Generating an M-File" on page 5-13

Opening the Pattern Search Tool

To open the tool, enter

```
psearchtool
```

at the MATLAB prompt. This opens the Pattern Search Tool, as shown in the following figure.



Defining a Problem in the Pattern Search Tool

You can define the problem you want to solve in the following fields:

- **Objective function** — The function you want to minimize. Enter a handle to an M-file function that computes the objective function. “Writing M-Files for

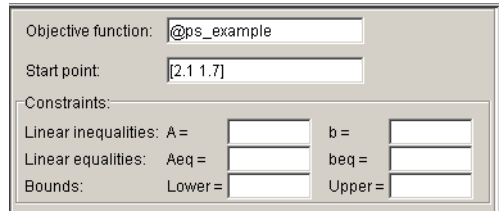
Functions You Want to Optimize" on page 1-3 describes how to write the M-file.

- **Start point** — The starting point for the pattern search algorithm

---

Note Do not use the Editor/Debugger to debug the M-file for the objective function while running the Pattern Search Tool. Doing so results in Java exception messages in the Command Window and makes debugging more difficult. Instead, call the objective function directly from the command line or pass it to the function `patternsearch`. To facilitate debugging, you can export your problem from the Pattern Search Tool to the MATLAB workspace, as described in "Importing and Exporting Options and Problems" on page 5-11.

The following figure shows these fields for the example described in "Example: Finding the Minimum of a Function" on page 3-6.



Constrained Problems

You can enter any constraints for the problem in the following fields in the **Constraints** pane:

- **Linear inequalities** — Enter the following for inequality constraints of the form $Ax \le b$:
  - Enter the matrix `A` in the **A =** field.
  - Enter the vector `b` in the **b =** fields.
- **Linear equalities** — Enter the following for equality constraints of the form $Aeq\ x = beq$:
  - Enter the matrix `Aeq` in the **Aeq =** field.
  - Enter the vector `beq` in the **beq =** field.

- **Bounds** — Enter the following information for bounds constraints of the form $lb \le x$ and $x \le ub$:
  - Enter the vector `lb` for the lower bound in the **Lower =** field.
  - Enter the vector `ub` in the **Upper =** field.

Leave the fields corresponding to constraints that do not appear in the problem empty.

Running a Pattern Search

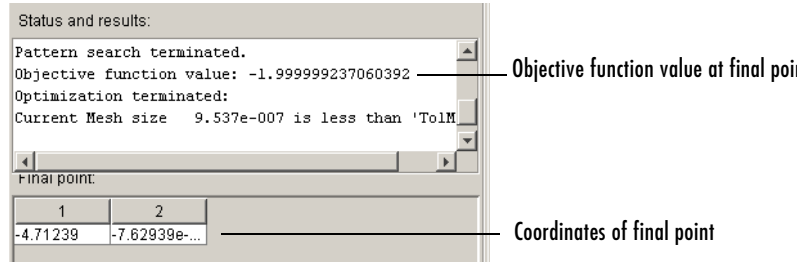
To run a pattern search, click **Start** in the **Run solver** pane. When you do so,

- The **Current iteration** field displays the number of the current iteration.
- The **Status and results** pane displays the message "`Pattern search running.`"

When the pattern search terminates, the **Status and results** pane displays

- The message "`Pattern search terminated.`"
- The objective function value at the final point
- The reason the pattern search terminated
- The coordinates of the final point

The following figure shows this information displayed when you run the example in "Example: Finding the Minimum of a Function" on page 3-6.



Example — A Constrained Problem

This section presents an example of performing a pattern search on a constrained minimization problem. The example minimizes the function

$$F(x) = \frac{1}{2}x^T Hx + f^T x$$

where

$$H = \begin{bmatrix} 36 & 17 & 19 & 12 & 8 & 15 \\ 17 & 33 & 18 & 11 & 7 & 14 \\ 19 & 18 & 43 & 13 & 8 & 16 \\ 12 & 11 & 13 & 18 & 6 & 11 \\ 8 & 7 & 8 & 6 & 9 & 8 \\ 15 & 14 & 16 & 11 & 8 & 29 \end{bmatrix}$$

$$f = \begin{bmatrix} 20 & 15 & 21 & 18 & 29 & 24 \end{bmatrix}$$

subject to the constraints

$$A \cdot x \le b$$
$$Aeq \cdot x = beq$$

where

$$A = \begin{bmatrix} -8 & 7 & 3 & -4 & 9 & 0 \end{bmatrix}$$

$$b = \begin{bmatrix} 7 \end{bmatrix}$$

$$Aeq = \begin{bmatrix} 7 & 1 & 8 & 3 & 3 & 3 \\ 5 & 0 & 5 & 1 & 5 & 8 \\ 2 & 6 & 7 & 1 & 1 & 8 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$beq = \begin{bmatrix} 84 & 62 & 65 & 1 \end{bmatrix}$$

Performing a Pattern Search on the Example

To perform a pattern search on the example, first enter

```
psearchtool
```

to open the Pattern Search Tool. Then set **Objective function** to

```
@lincontest7
```

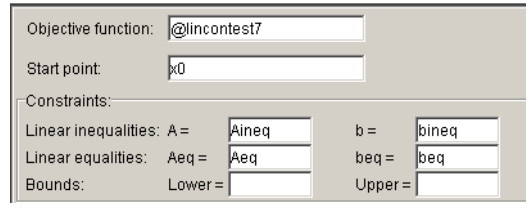
an M-file included in the toolbox that computes the objective function for the example. Because the matrices and vectors defining the starting point and constraints are large, it is more convenient to set their values as variables in the MATLAB workspace first and then enter the variable names in the Pattern Search Tool. To do so, enter

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = [7];
Aeq = [7 1 8 3 3 3; 5 0 5 1 5 8; 2 6 7 1 1 8; 1 0 0 0 0 0];
beq = [84 62 65 1];
```

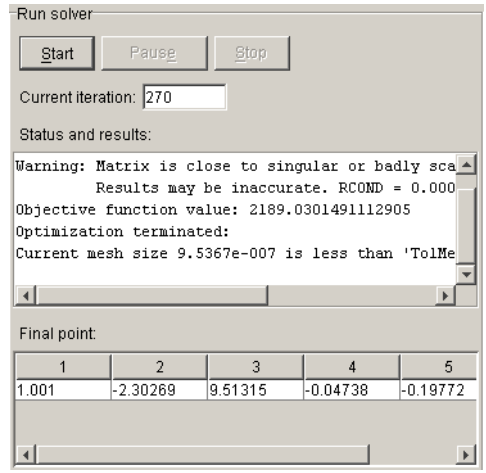
Then, enter the following in the Pattern Search Tool:

- Set **Initial point** to `x0`.
- Set the following **Linear inequalities**:
  - Set **A =** to `Aineq`.
  - Set **b =** to `bineq`.
  - Set **Aeq =** to `Aeq`.
  - Set **beq =** to `beq`.

The following figure shows these settings in the Pattern Search Tool.



Then click **Start** to run the pattern search. When the search is finished, the results are displayed in **Status and results** pane, as shown in the following figure.



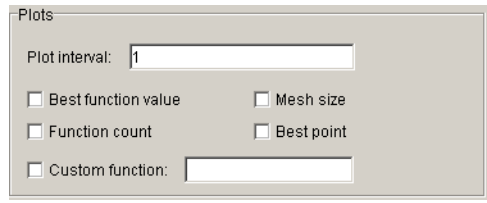
Pausing and Stopping the Algorithm

While pattern search is running, you can

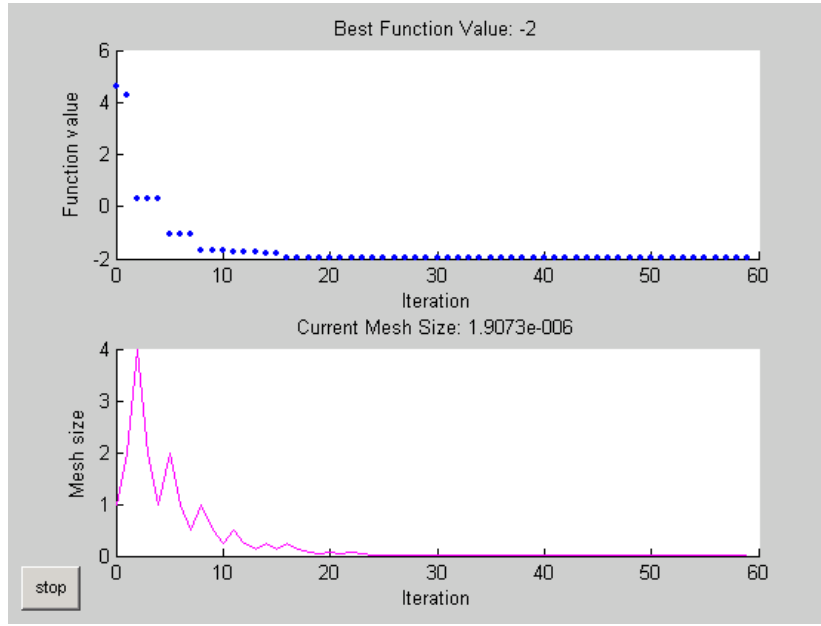
- Click **Pause** to temporarily suspend the algorithm. To resume the algorithm from the point at which you paused, click **Resume**.
- Click **Stop** to stop the algorithm. The **Status and results** pane displays the objective function value of the current point at the moment you clicked **Stop**.

Displaying Plots

The **Plots** pane, shown in the following figure, enables you to display various plots of the results of a pattern search.



Select the check boxes next to the plots you want to display. For example, if you select **Best function value** and **Mesh size**, and run the example described in "Example: Finding the Minimum of a Function" on page 3-6, the tool displays the plots shown in the following figure.



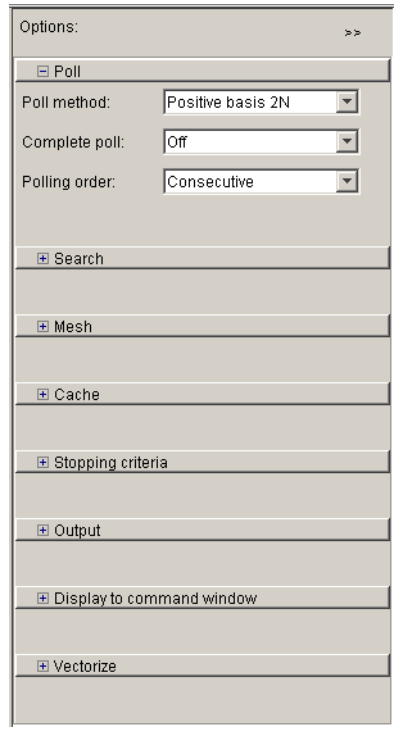
The upper plot displays the objective function value at each iteration. The lower plot displays the coordinates of the point with the best objective function value at the current iteration.

Note When you display more than one plot, clicking on any plot displays a larger version of it in a separate window.

"Plot Options" on page 6-4 describes the types of plots you can create.

Setting Options in the Pattern Search Tool

You can set options for a pattern search in the **Options** pane, shown in the figure below.



For a detailed description of the available options, see “Pattern Search Options” on page 6-21.

Setting Options as Variables in the MATLAB workspace

You can set numerical options either directly, by typing their values in the corresponding edit box, or by entering the name of a variable in the MATLAB workspace that contains the option values. For options whose values are large matrices or vectors, it is often more convenient to define their values as variables in the MATLAB workspace.

Importing and Exporting Options and Problems

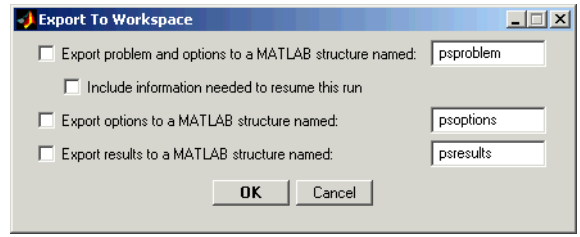
You can export options and problem structures from the Pattern Search Tool to the MATLAB workspace, and later import them in a subsequent session of the tool. This provides an easy way to save your work for future sessions of the Pattern Search Tool. The following sections describe how to import and export options and problem structures.

Exporting Options, Problems, and Results

After working on a problem using the Pattern Search Tool, you can export the following information to the MATLAB workspace:

- The problem definition, including
  - The objective function
  - The start point
  - Constraints on the problem
- The current options
- The results of the algorithm

To do so, click the **Export** button or select **Export to Workspace** from the **File** menu. This opens the dialog box shown in the following figure.



The dialog provides the following options:

- To save the objective function and options in a MATLAB structure, select **Export problem and options to a MATLAB structure named** and enter a name for the structure.

  If you have run a pattern search in the current session and you select **Include information needed to resume this run**, the final point from the

last search is saved in place of **Start point**. Use this option if you want to run the pattern search at a later time from the final point of the last search.

See "Importing a Problem" on page 5-13.

- To save only the options, select **Export options to a MATLAB structure named** and enter a name for the options structure.
- To save the results of the last run of the algorithm, select **Export results to a MATLAB structure named** and enter a name for the results structure.

Example — Running patternsearch on an Exported Problem

To export the problem described in "Example — A Constrained Problem" on page 5-6 and perform a pattern search on it using the function `patternsearch` at the command line, do the following steps:

1. Click **Export to Workspace**.
2. In the **Export to Workspace** dialog box, enter a name for the problem structure, such as `my_psroblem`, in the **Export problems and options to a MATLAB structure named** field.
3. Call the function `patternsearch` with `my_psproblem` as the input argument.

   ```
   [x fval] = patternsearch(my_psproblem)
   ```

   This returns

   ```
   x =

       1.0010   -2.3027    9.5131   -0.0474   -0.1977    1.3083


   fval =

     2.1890e+003
   ```

See "Performing a Pattern Search from the Command Line" on page 5-14 for form information.

Importing Options

To import an options structure for a pattern search from the MATLAB workspace, select **Import Options** from the **File** menu. This opens a dialog box

that displays a list of the valid pattern search options structures in the MATLAB workspace. When you select an options structure and click **Import**, the Pattern Search Tool resets its options to the values in the imported structure.

Note You cannot import options structures that contain any invalid option fields. Structures with invalid fields are not displayed in the **Import Pattern Search Options** dialog box.

You can create an options structure in either of the following ways:

- Calling `psoptimset` with `options` as the output
- By saving the current options from the **Export to Workspace** dialog box in the Pattern Search Tool

Importing a Problem

To import a problem that you previously exported from the Pattern Search Tool, select **Import Problem** from the **File** menu. This opens the dialog box that displays a list of the pattern search problem structures in the MATLAB workspace. When you select a problem structure and click **OK**, the Pattern Search Tool resets the problem definition and the options to the values in the imported structure. In addition, if you selected **Include information needed to resume this run** when you created the problem structure, the tool resets **Start point** to the final point of the last run prior to exporting the structure.

See “Exporting Options, Problems, and Results” on page 5-11.

Generating an M-File

To create an M-file that runs a pattern search using the objective function and options you specify in the Pattern Search Tool, select **Generate M-File** from the **File** menu and save the M-file in a directory on the MATLAB path. Calling this M-file at the command line returns the same results as the Pattern Search Tool, using the fitness function and options settings that were in place when you generated the M-file.

Performing a Pattern Search from the Command Line

As an alternative to using the Pattern Search Tool, you can call the function `patternsearch` at the command line. This section explains how to do so and covers the following topics:

- "Calling patternsearch with the Default Options" on page 5-14
- "Setting Options for patternsearch at the Command Line" on page 5-16
- "Using Options and Problems from the Pattern Search Tool" on page 5-18

Calling patternsearch with the Default Options

This section describes how to perform a pattern search with the default options.

Pattern Search on Unconstrained Problems

For an unconstrained problem, call `patternsearch` with the syntax

```
[x fval] = patternsearch(@objectfun, x0)
```

The output arguments are

- `x` — The final point
- `fval` — The value of the objective function at `x`

The required input arguments are

- `@objectfun` — A function handle to the objective function `objectfun`, which you can write as an M-file. See "Writing M-Files for Functions You Want to Optimize" on page 1-3 to learn how to do this.
- `x0` — The initial point for the pattern search algorithm

As an example, you can run the example described in "Example: Finding the Minimum of a Function" on page 3-6 from the command line by entering

```
[x fval] = patternsearch(@ps_example, [2.1 1.7])
```

This returns

```
Optimization terminated:
Current mesh size 9.5367e-007 is less than 'TolMesh'.
```

```
x =

   -4.7124   -0.0000


fval =

   -2.0000
```

Pattern Search on Constrained Problems

If your problem has constraints, use the syntax

```
[x fval] = patternsearch(@objfun, x0, A, b Aeq, beq, lb, ub)
```

where

- `A` is a matrix and `b` is vector that represent inequality constraints of the form $Ax \le b$.
- `Aeq` is a matrix and `beq` is a vector that represent equality constraints of the form $Aeq\,x = beq$.
- `lb` and `ub` are vectors representing bound constraints of the form $lb \le x$ and $x \le ub$, respectively.

You only need to pass in the constraints that are part of the problem. For example, if there are no bound constraints, use the syntax

```
[x fval] = patternsearch(@objfun, x0, A, b Aeq, beq)
```

Use empty brackets `[]` for constraint arguments that are not needed for the problem. For example, if there are no inequality constraints, use the syntax

```
[x fval] = patternsearch(@objfun, x0, [], [], Aeq, beq, lb, ub)
```

Additional Output Arguments

To get more information about the performance of the pattern search, you can call `patternsearch` with the syntax

```
[x fval exitflag output] = patternsearch(@objfun, x0)
```

Besides `x` and `fval`, this returns the following additional output arguments:

- `exitflag` — Integer indicating whether the algorithm was successful

- output — Structure containing information about the performance of the solver

See the reference page for patternsearch for more information about these arguments.

Setting Options for patternsearch at the Command Line

You can specify any of the options that are available in the Pattern Search Tool by passing an options structure as an input argument to patternsearch using the syntax

```
[x fval] = patternsearch(@fitnessfun, nvars, ...
            A, b, Aeq, beq, lb, ub, options)
```

Pass in empty brackets [] for any constraints that do not appear in the problem.

You create the options structure using the function psoptimset.

```
options = psoptimset
```

This returns the options structure with the default values for its fields.

```
options =

            TolMesh: 1.0000e-006
               TolX: 1.0000e-006
             TolFun: 1.0000e-006
            TolBind: 1.0000e-003
            MaxIter: '100*numberofvariables'
        MaxFunEvals: '2000*numberofvariables'
    MeshContraction: 0.5000
      MeshExpansion: 2
    MeshAccelerator: 'off'
         MeshRotate: 'on'
    InitialMeshSize: 1
          ScaleMesh: 'on'
        MaxMeshSize: Inf
         PollMethod: 'positivebasis2n'
       CompletePoll: 'off'
       PollingOrder: 'consecutive'
```

```
      SearchMethod: []
    CompleteSearch: 'off'
           Display: 'final'
        OutputFcns: []
          PlotFcns: []
      PlotInterval: 1
             Cache: 'off'
         CacheSize: 10000
          CacheTol: 2.2204e-016
        Vectorized: 'off'
```

The function `patternsearch` uses these default values if you do not pass in `options` as an input argument.

The value of each option is stored in a field of the `options` structure, such as `options.MeshExpansion`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the mesh expansion factor for the pattern search, enter

```
options.MeshExpansion

ans =

    2
```

To create an `options` structure with a field value that is different from the default, use the function `psoptimset`. For example, to change the mesh expansion factor to `3` instead of its default value `2`, enter

```
options = psoptimset('MeshExpansion', 3)
```

This creates the `options` structure with all values set to their defaults except for `MeshExpansion`, which is set to `3`.

If you now call `patternsearch` with the argument `options`, the pattern search uses a mesh expansion factor of `3`.

If you subsequently decide to change another field in the `options` structure, such as setting `PlotFcns` to `@psplotmeshsize`, which plots the mesh size at each iteration, call `psoptimset` with the syntax

```
options = psoptimset(options, 'PlotFcns', @psplotmeshsize)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@plotmeshsize`. Note that if you omit the `options` input argument, `psoptimset` resets `MeshExpansion` to its default value, which is `2.0`.

You can also set both `MeshExpansion` and `PlotFcns` with the single command

```
options = psoptimset('MeshExpansion',3,'PlotFcns',@plotmeshsize)
```

Using Options and Problems from the Pattern Search Tool

As an alternative to creating the options structure using `psoptimset`, you can set the values of options in the Pattern Search Tool and then export the options to a structure in the MATLAB workspace, as described in "Exporting Options, Problems, and Results" on page 5-11. If you export the default options in the Pattern Search Tool, the resulting `options` structure has the same settings as the default structure returned by the command

```
options = psoptimset
```

except for the default value of `'Display'`, which is `'final'` when created by `psoptimset`, but `'none'` when created in the Pattern Search Tool.

You can also export an entire problem from the Pattern Search Tool and run it from the command line. See "Example — Running patternsearch on an Exported Problem" on page 5-12 for an example.

Pattern Search Examples

This section explains how to set options for a pattern search.

- "Poll Method" on page 5-19
- "Complete Poll" on page 5-21
- "Using a Search Method" on page 5-25
- "Mesh Expansion and Contraction" on page 5-28
- "Mesh Accelerator" on page 5-33
- "Using Cache" on page 5-34
- "Setting Tolerances for the Solver" on page 5-36

Poll Method

At each iteration, a pattern search polls the points in the current mesh — that is, it computes the objective function at the mesh points to see if there is one whose function value is less than the function value at the current point. "How Pattern Search Works" on page 3-13 provides an example of polling. You can specify the pattern that defines the mesh by the **Poll method** option. The default pattern, `Positive basis 2N`, consists of the following 2N directions, where N is the number of independent variables for the objective function.

[100…0]
[010…0]
…
[000…1]
[–1 00…0]
[0 –10…0]
…
[0 00…–1]

For example, if objective function has three independent variables, the `Positive basis 2N`, consists of the following six vectors.

$$\begin{bmatrix}1 & 0 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$$

$$\begin{bmatrix}-1 & 0 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & -1 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 0 & -1\end{bmatrix}$$

Alternatively, you can set **Poll method** to `Positive basis NP1`, the pattern consisting of the following N + 1 directions.

[100…0]
[010…0]
…
[000…1]
[–1 –1–1…–1]

For example, if objective function has three independent variables, the `Positive basis Np1`, consists of the following four vectors.

$$\begin{bmatrix}1 & 0 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$$

$$\begin{bmatrix}-1 & -1 & -1\end{bmatrix}$$

A pattern search will sometimes run faster using `Positive basis Np1` as the **Poll method**, because the algorithm searches fewer points at each iteration. For example, if you run a pattern search on the example described in "Example — A Constrained Problem" on page 5-6, the algorithm performs 2080 function evaluations with `Positive basis 2N`, the default **Poll method**, but only 1413 function evaluations using `Positive basis 2P1`.

However, if the objective function has many local minima, using `Positive basis 2N` as the **Poll method** might avoid finding a local minimum that is not the global minimum, because the search explores more points around the current point at each iteration.

Complete Poll

By default, if the pattern search finds a mesh point that improves the value of the objective function, it stops the poll and sets that point as the current point for the next iteration. When this occurs, some mesh points might not get polled. Some of these unpolled points might have an objective function value that is even lower than the first one the pattern search finds.

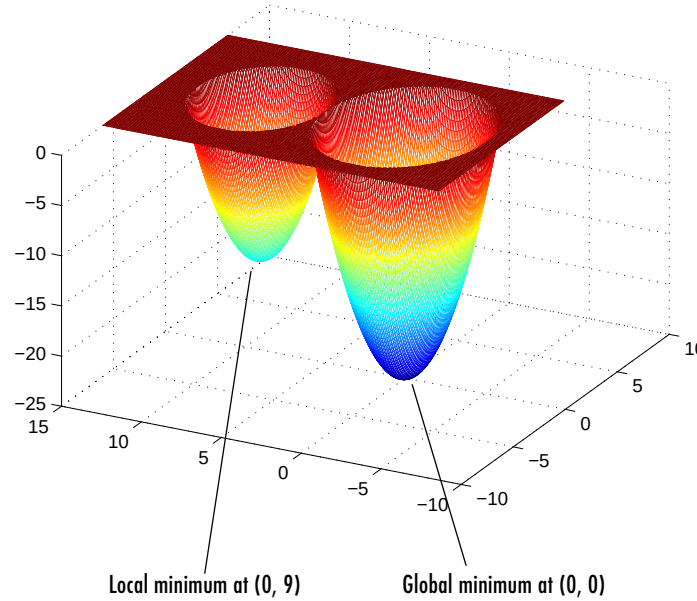
For problems in which there are several local minima, it is sometimes preferable to make the pattern search poll *all* the mesh points at each iteration and choose the one with the best objective function value. This enables the pattern search to explore more points at each iteration and thereby potentially avoid a local minimum that is not the global minimum. You can make the pattern search poll the entire mesh setting **Complete poll** to `On` in **Poll** options.

Example — Using a Complete Poll in a Pattern Search

As an example, consider the following function.

$$f(x_1, x_2) = \begin{cases} x_1^2 + x_2^2 - 25 & \text{for } x_1^2 + x_2^2 \le 25 \\ x_1^2 + (x_2 - 9)^2 - 16 & \text{for } x_1^2 + (x_2 - 9)^2 \le 16 \\ 0 & \text{otherwise} \end{cases}$$

The following figure shows a plot of the function.



The global minimum of the function occurs at (0, 0), where its value is -25. However, the function also has a local minimum at (0, 9), where its value is -16.

To create an M-file that computes the function, copy and paste the following code into a new M-file in the MATLAB Editor.

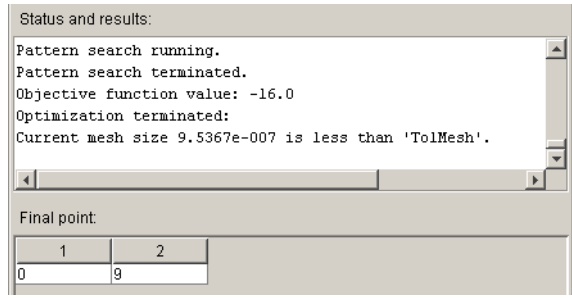
```
function z = poll_example(x)
if x(1)^2 + x(2)^2 <= 25
    z = x(1)^2 + x(2)^2 - 25;
elseif x(1)^2 + (x(2) - 9)^2 <= 16
    z = x(1)^2 + (x(2) - 9)^2 - 16;
else z = 0;
end
```

Then save the file as `poll_example.m` in a directory on the MATLAB path.

To run a pattern search on the function, enter the following in the Pattern Search Tool:

- Set **Objective function** to `@poll_example`.
- Set **Start point** to `[0 5]`.
- Set **Level of display** to `Iterative` in **Display to command window** options.

Click **Start** to run the pattern search with **Complete poll** set to `Off`, its default value. The Pattern Search Tool displays the results in the **Status and results** pane, as shown in the following figure.



The pattern search returns the local minimum at (0, 9). At the initial point, (0, 5), the objective function value is 0. At the first iteration, the search polls the following mesh points.

$f((0, 5) + (1, 0)) = f(1, 5) = 0$

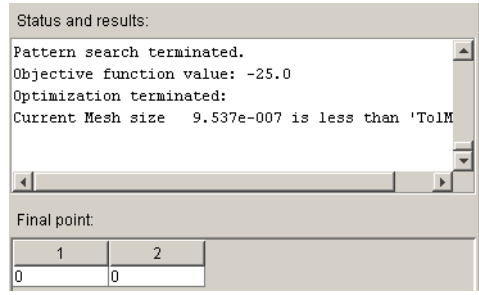
$f((0, 5) + (0, 1)) = f(0, 6) = -7$

As soon as the search polls the mesh point (0, 6), at which the objective function value is less than at the initial point, it stops polling the current mesh and sets the current point at the next iteration to (0, 6). Consequently, the search moves toward the local minimum at (0, 9) at the first iteration.You see this by looking at the first two lines of the command line display.

```
Iter     f-count        MeshSize      f(x)        Method
    0        1              1            0     Start iterations
    1        3              2           -7     Successful Poll
```

Note that the pattern search performs only two evaluations of the objective function at the first iteration, increasing the total function count from 1 to 3.

Next, set **Complete poll** to `On` and click **Start**. The **Status and results** pane displays the following results.



This time, the pattern search finds the global minimum at (0, 0). The difference between this run and the previous one is that with **Complete poll** set to `On`, at the first iteration the pattern search polls all four mesh points.

$f((0, 5) + (1, 0)) = f(1, 5) = 0$

$f((0, 5) + (0, 1)) = f(0, 6) = -6$

$f((0, 5) + (-1, 0)) = f(-1, 5) = 0$

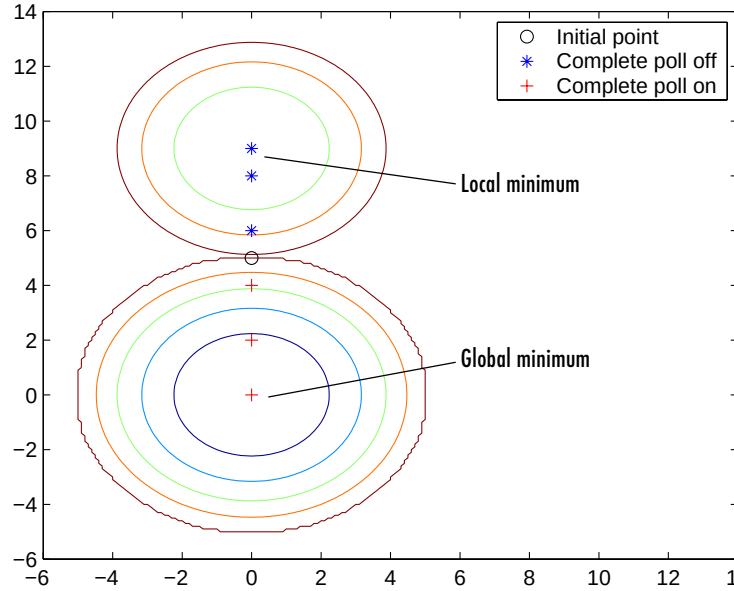
$f((0, 5) + (0, -1)) = f(0, 4) = -9$

Because the last mesh point has the lowest objective function value, the pattern search selects it as the current point at the next iteration. The first two lines of the command-line display show this.

```
Iter     f-count        MeshSize      f(x)        Method
    0        1              1            0     Start iterations
    1        5              2           -9     Successful Poll
```

In this case, the objective function is evaluated four times at the first iteration. As a result, the pattern search moves toward the global minimum at (0, 0).

The following figure compares the sequence of points returned when **Complete poll** is set to `Off` with the sequence when **Complete poll** is `On`.



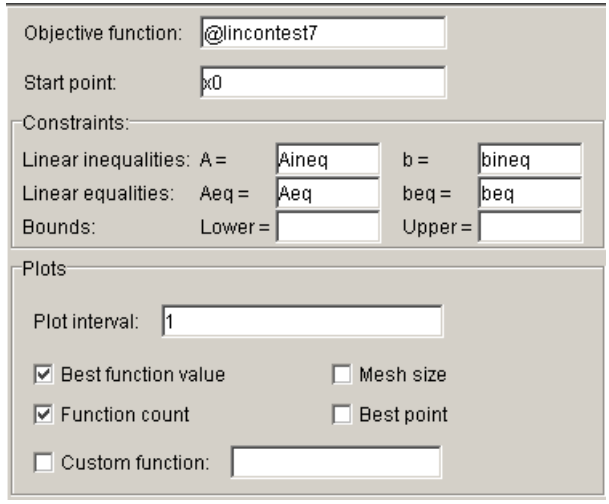
Using a Search Method

In addition to polling the mesh points, the pattern search algorithm can perform an optional step at every iteration, called *search*. At each iteration, the search step applies another optimization method to the current point. If this search does not improve the current point, the poll step is performed.

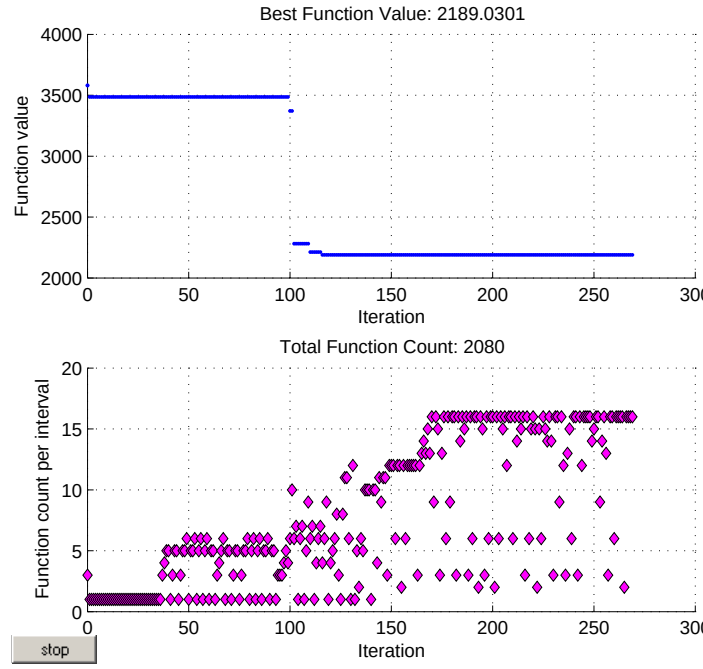
The following example illustrates the use of a search method on the problem described in “Example — A Constrained Problem” on page 5-6. To set up the example, enter the following commands at the MATLAB prompt to define the initial point and constraints.

```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0 ];
bineq = [7];
Aeq = [7 1 8 3 3 3; 5 0 5 1 5 8; 2 6 7 1 1 8; 1 0 0 0 0 0];
beq = [84 62 65 1];
```

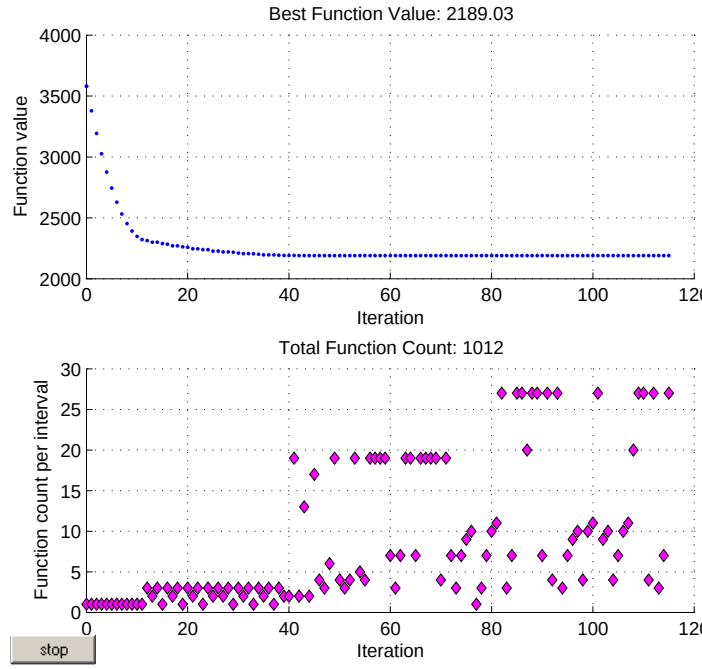
Then enter the settings shown in the following figure in the Pattern Search Tool.



For comparison, click **Start** to run the example without a search method. This displays the plots shown in the following figure.



To see the effect of using a search method, select `Positive Basis Np1` in the **Search method** field in **Search** options. This sets the search method to be a pattern search using the pattern for `Positive basis Np1`. Then click **Start** to run the genetic algorithm. This displays the following plots.

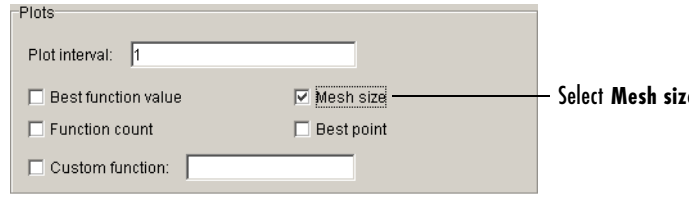


Note that using the search method reduces the total function count — the number of times the objective function was evaluated — by almost 50 percent, and reduces the number of iterations from 270 to 120.

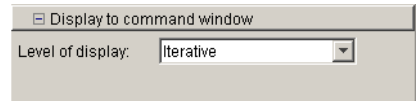
Mesh Expansion and Contraction

The **Expansion factor** and **Contraction factor** options, in **Mesh** options, control how much the mesh size is expanded or contracted at each iteration. With the default **Expansion factor** value of `2`, the pattern search multiplies the mesh size by `2` after each successful poll. With the default **Contraction factor** value of `0.5`, the pattern search multiplies the mesh size by `0.5` after each unsuccessful poll.

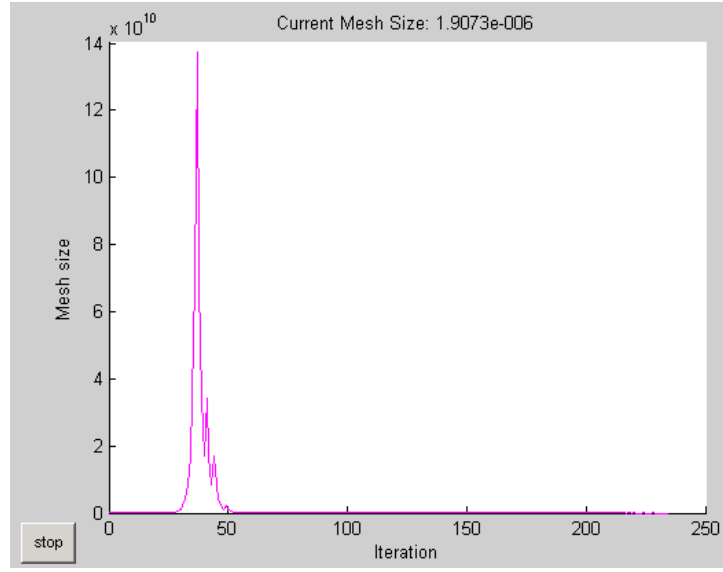
You can view the expansion and contraction of the mesh size during the pattern search by selecting **Mesh size** in the **Plots** pane.



To also display the values of the mesh size and objective function at the command line, set **Level of Display** to `Iterative` in the **Display to command window** options.



When you run the example described in "Example — A Constrained Problem" on page 5-6, the Pattern Search Tool displays the following plot.



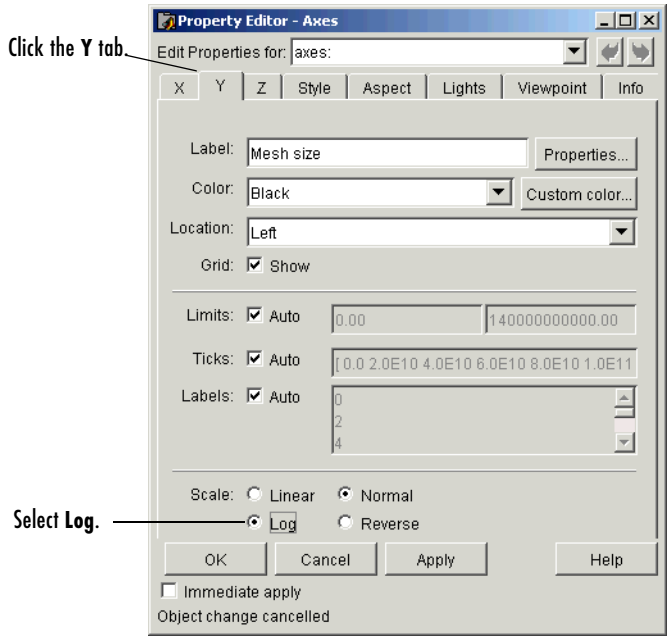
To see the changes in mesh size more clearly, change the *y*-axis to logarithmic scaling as follows:

1 Select **Axes Properties** from the **Edit** menu in the plot window.

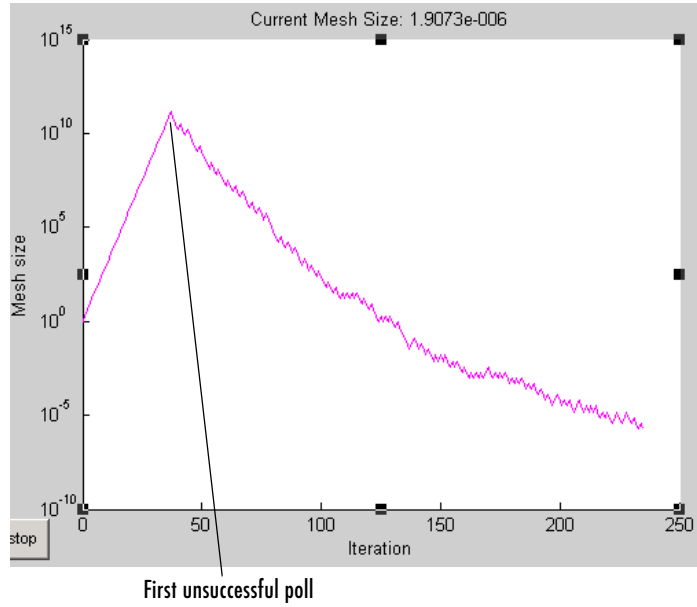
2 In the Properties Editor, select the **Y** tab.

3 Set **Scale** to **Log**.

The following figure shows these settings in the Property Editor.



When you click **OK**, the plot appears as shown in the following figure.



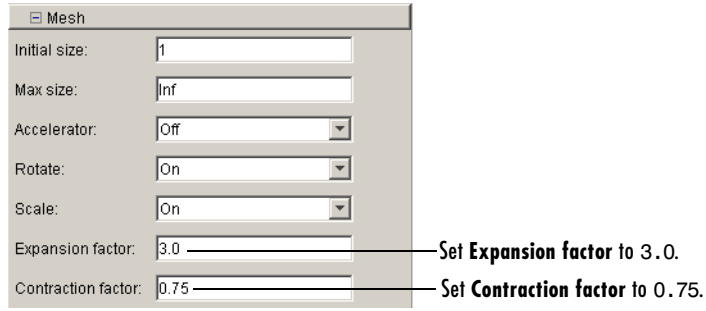
The first 37 iterations result in successful polls, so the mesh sizes increase steadily during this time. You can see that the first unsuccessful poll occurs at iteration 38 by looking at the command-line display for that iteration.

```
36        39      6.872e+010          3486     Successful Poll
37        40      1.374e+011          3486     Successful Poll
38        43      6.872e+010          3486     Refine Mesh
```

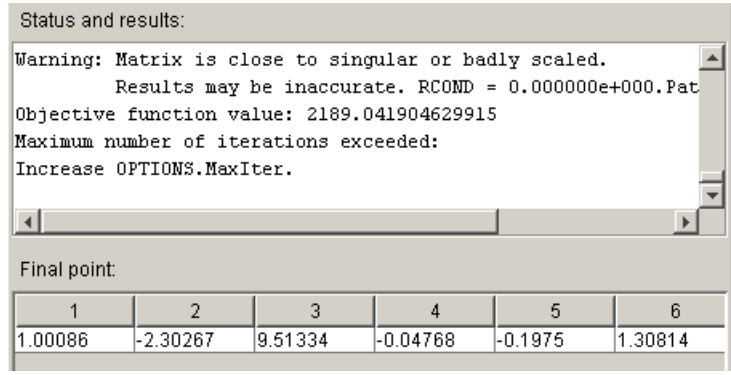
Note that at iteration 37, which is successful, the mesh size doubles for the next iteration. But at iteration 38, which is unsuccessful, the mesh size is multiplied `0.5`.

To see how **Expansion factor** and **Contraction factor** affect the pattern search, make the following changes:

- Set **Expansion factor** to `3.0`.
- Set **Contraction factor** to `0.75`.

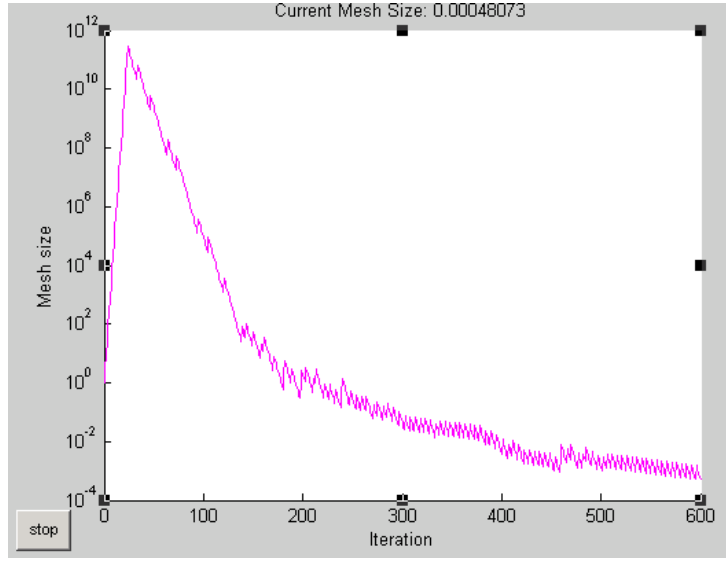


Then click **Start**. The **Status and results** pane shows that the final point is approximately the same as with the default settings of **Expansion factor** and **Contraction factor**, but that the pattern search takes longer to reach that point.



The algorithm halts because it exceeds the maximum number of iterations, whose value you can set in the **Max iteration** field in the **Stopping criteria** options. The default value is 100 times the number of variables for the objective function, which is 6 in this example.

When you change the scaling of the *y*-axis to logarithmic, the mesh size plot appears as shown in the following figure.



Note that the mesh size increases faster with **Expansion factor** set to `3.0`, as compared with the default value of `2.0`, and decreases more slowly with **Contraction factor** set to `0.75`, as compared with the default value of `0.5`.

Mesh Accelerator

The mesh accelerator can make a pattern search converge faster to the optimal point by reducing the number of iterations required to reach the mesh tolerance. When the mesh size is below a certain value, the pattern search contracts the mesh size by a factor smaller than the **Contraction factor** factor.

Note We recommend that you only use the mesh accelerator for problems in which the objective function is not too steep near the optimal point, or you might lose some accuracy. For differentiable problems, this means that the absolute value of the derivative is not too large near the solution.

To use the mesh accelerator, set **Accelerator** to `On` in **Mesh** options. When you run the example describe in "Example — A Constrained Problem" on page 5-6, the number of iterations required to reach the mesh tolerance is 246, as compared with 270 when **Accelerator** is set to `Off`.

You can see the effect of the mesh accelerator by setting **Level of display** to **Iterative** in the **Display to command window**. Run the example with **Accelerator** set to `On`, and then run it again with **Accelerator** set to `Off`. The mesh sizes are the same until iteration 226, but differ at iteration 227. The MATLAB Command Window displays the following lines for iterations 226 and 227 with **Accelerator** set to `Off`.

```
Iter     f-count        MeshSize      f(x)        Method
226     1501     6.104e-005          2189     Refine Mesh
227     1516     3.052e-005          2189     Refine Mesh
```

Note that the mesh size is multiplied by `0.5`, the default value of **Contraction factor factor**.

For comparison, the Command Window displays the following lines for the same iteration numbers with **Accelerator** set to `On`.

```
Iter     f-count        MeshSize      f(x)        Method
226     1501     6.104e-005          2189     Refine Mesh
227     1516     1.526e-005          2189     Refine Mesh
```

In this case the mesh size is multiplied by `0.25`.

Using Cache

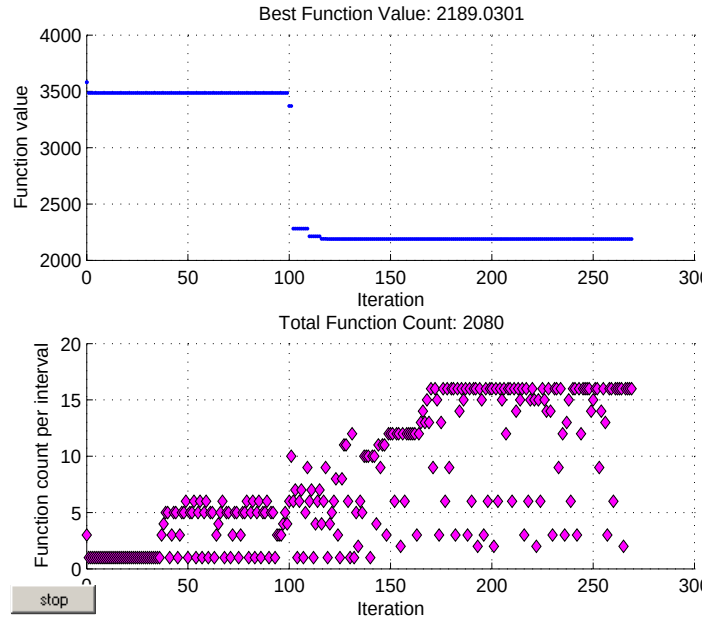
Typically, at any given iteration of a pattern search, some of the mesh points might coincide with mesh points at previous iterations. By default, the pattern search recomputes the objective function at these mesh points even though it has already computed their values and found that they are not optimal. If computing the objective function takes a long time — say, several minutes — this can make the pattern search run significantly longer.

You can eliminate these redundant computations by using a cache, that is, by storing a history of the points that the pattern search has already visited. To do so, set **Cache** to `On` in **Cache** options. At each poll, the pattern search checks to see whether the current mesh point is within a specified tolerance, **Tolerance**, of a point in the cache. If so, the search does not compute the

objective function for that point, but uses the cached function value and moves on to the next point.

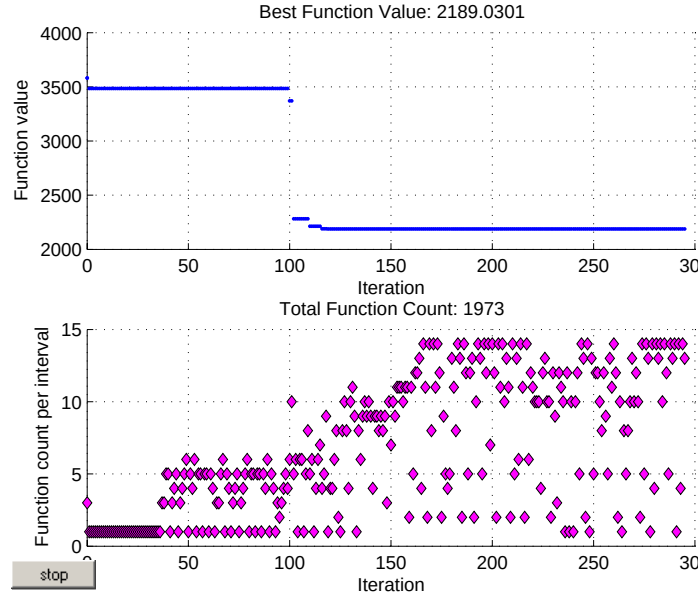
Note When **Cache** is set to `On`, the pattern search might fail to identify a point in the current mesh that improves the objective function because it is within the specified tolerance of a point in the cache. As a result, the pattern search might run for more iterations with **Cache** set to `On` than with **Cache** set to `Off`. It is generally a good idea to keep the value of **Tolerance** very small, especially for highly nonlinear objective functions.

To illustrate this, select **Best function value** and **Function count** in the **Plots** pane and run the example described in "Example — A Constrained Problem" on page 5-6 with **Cache** set to `Off`. After the pattern search finishes, the plots appear as shown in the following figure.



Note that the total function count is 2080.

Now, set **Cache** to `On` and run the example again. This time, the plots appear as shown in the following figure.



This time, the total function count is reduced to 1973.

Setting Tolerances for the Solver

Tolerance refers to how small a parameter, such a mesh size, can become before the search is halted or changed in some way. You can specify the value of the following tolerances:

- **Mesh tolerance** — When the current mesh size is less than the value of **Mesh tolerance**, the algorithm halts.
- **X tolerance** — After a successful poll, if the distance from the previous best point to the current best point is less than the value of **X tolerance**, the algorithm halts.
- **Function tolerance** — After a successful poll, if the distance from the previous best point to the current best point is less than the value of **X tolerance**, the algorithm halts.

- **Bind tolerance** — Bind tolerance applies to constrained problems and specifies how close a point must get to the boundary of the feasible region before a linear constraint is considered to be active. When a linear constraint is active, the pattern search polls points in directions parallel to the linear constraint boundary as well as the mesh points.

  Usually, you should set **Bind tolerance** to be at least as large as the maximum of **Mesh tolerance**, **X tolerance**, and **Function tolerance**.

Example — Setting Bind Tolerance

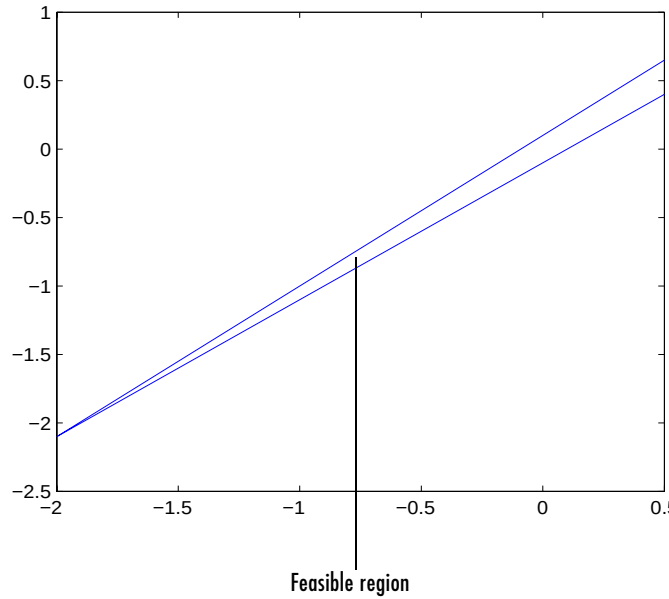
The following example illustrates of how **Bind tolerance** affects a pattern search. The example finds the minimum of

$$f(x_1, x_2) = \sqrt{x_1^2 + x_2^2}$$

subject to the constraints

$$-11x_1 + 10x_2 \le 10$$
$$10x_1 - 10x_2 \le 10$$

Note that you can compute the objective function using the function `norm`. The feasible region for the problem lies between the two lines in the following figure.



Running a Pattern Search with the Default Bind Tolerance

To run the example, enter `psearchtool` to open the Pattern Search Tool and enter the following information:

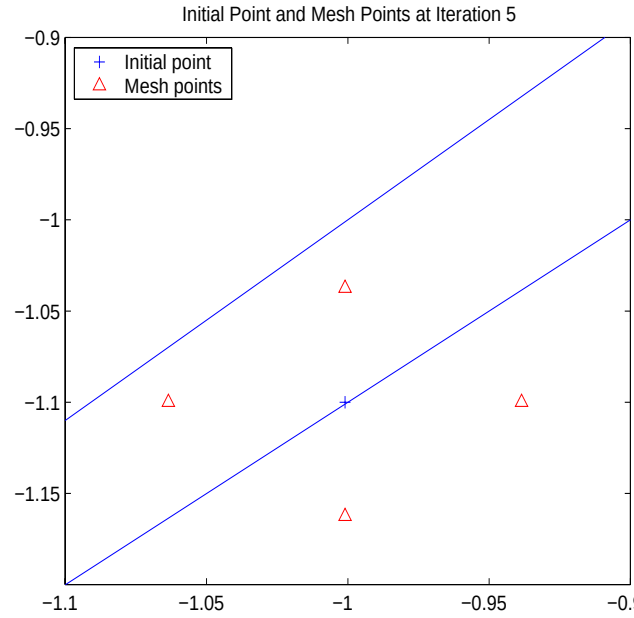
- Set **Objective function** to `@(x) norm(x)`.
- Set **Start point** to `[-1.001 -1.1]`.
- Select **Mesh size** in the `Plots` pane.
- Set **Level of display** to `Iterative` in the **Display to command window** options.

Then click **Start** to run the pattern search.

The display in the MATLAB Command Window shows that the first four polls are unsuccessful, because the mesh points do not lie in the feasible region.

Iter	f-count	MeshSize	f(x)	Method
0	1	1	1.487	Start iterations
1	1	0.5	1.487	Refine Mesh
2	1	0.25	1.487	Refine Mesh
3	1	0.125	1.487	Refine Mesh
4	1	0.0625	1.487	Refine Mesh

The pattern search contracts the mesh at each iteration until one of the mesh points lies in the feasible region. The following figure shows a close-up of the initial point and mesh points at iteration 5.



The top mesh point, which is (-1.001, -1.0375), has a smaller objective function value than the initial point, so the poll is successful.

Because the distance from the initial point to lower boundary line is less than the default value of **Bind tolerance**, which is `0.0001`, the pattern search does

not consider the linear constraint $10x_1 - 10x_2 \le 10$ to be active, so it does not search points in a direction parallel to the boundary line.

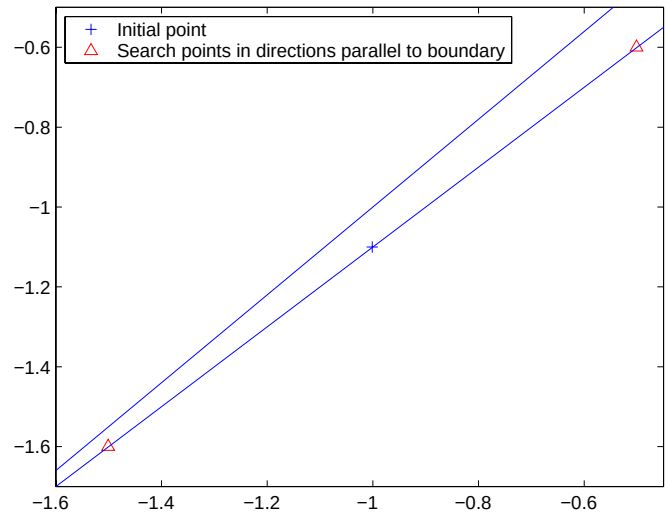
Increasing the Value of Bind Tolerance

To see the effect of bind tolerance, change **Bind tolerance** to `0.01` and run the pattern search again.

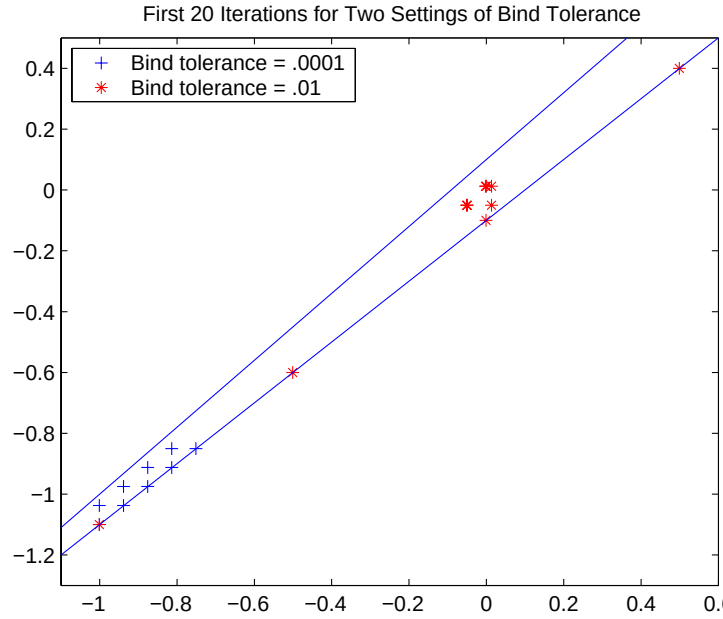
This time, the display in the MATLAB Command Window shows that the first two iterations are successful.

```
Iter     f-count         MeshSize       f(x)        Method
    0           1              1        1.487     Start iterations
    1           2              2       0.7817     Successful Poll
    2           3              4       0.6395     Successful Poll
```

Because the distance from the initial point to the boundary is less than **Bind tolerance**, the second linear constraint is active. In this case, the pattern search polls points in directions parallel to the boundary line $10x_1 - 10x_2 = 10$, resulting in successful poll. The following figure shows the initial point with two addition search points in directions parallel to the boundary.



The following figure compares the sequences of points during the first 20 iterations of the pattern search for both settings of **Bind tolerance**.



Note that when **Bind tolerance** is set to `.01`, the points move toward the optimal point more quickly. The pattern search requires only 90 iterations. When **Bind tolerance** is set to `.0001`, the search requires 124 iterations. However, when the feasible region does not contain very acute angles, as it does in this example, increasing **Bind tolerance** can increase the number of iterations required, because the pattern search tends to poll more points.

Parameterizing Functions Called by patternsearch or ga

Sometimes you might want to write functions that are called by `patternsearch` or `ga`, which have additional parameters besides the independent variable. For example, suppose you want to minimize the following function:

$$f(x) = (a - bx_1^2 + x_1^{4/3})x_1^2 + x_1x_2 + (-c + cx_2^2)x_2^2$$

for different values of a, b, and c. Because `patternsearch` and `ga` accept objective or fitness functions that depend only on x, you must provide the additional parameters a, b, and c to the function before calling `patternsearch` or `ga`. The following sections describe two ways to do this:

- "Parameterizing Functions Using Anonymous Functions" on page 5-42
- "Parameterizing a Function Using a Nested Function" on page 5-44

The examples in these sections show how to parameterize the objective function, but you can use the same methods to parameterize any user-defined functions called by `patternsearch` or `ga`, for example a custom search method for `patternsearch` or a custom scaling function for `ga`.

Parameterizing Functions Using Anonymous Functions

To parameterize your function, first write an M-file containing the following code:

```
function y = parameterfun(x,a,b,c)
y = (a - b*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + ...

          (-c + c*x(2)^2)*x(2)^2;
```

Save the M- file as `myfun.m` in a directory on the MATLAB path.

Now, suppose you want to minimize the function for the parameter values a = 4, b =2.1, and c = 4. To do so, define a function handle to an anonymous function by entering the following commands at the MATLAB prompt:

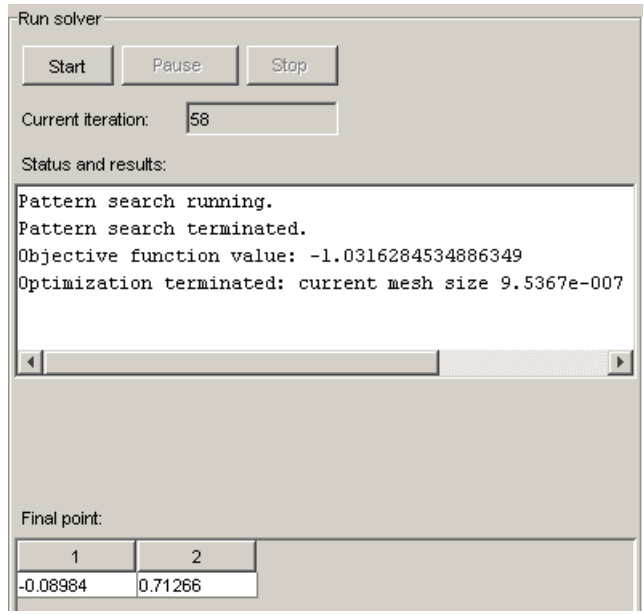
```
a = 4; b = 2.1; c = 4;    % Define parameter values
objfun = @(x) parameterfun(x,a,b,c);
x0 = [0.5 0.5];
```

If you are using the pattern search tool,

- Set **Objective function** to `objfun`.
- Set **Start point** to `x0`.

The following figure shows these settings in the Pattern Search Tool.

Next, click **Start** to run the optimization. The Status and results pane displays the final answer.



If you subsequently decide to change the values of `a`, `b`, and `c`, you must recreate the anonymous function. For example,

```
a = 3.6; b = 2.4; c = 5;    % Define parameter values
objfun = @(x) parameterfun(x,a,b,c);
```

Parameterizing a Function Using a Nested Function

As an alternative to parameterizing the objective function as an anonymous function, you can write a single M-file that

- Accepts `a`, `b`, `c`, and `x0` as inputs.
- Contains the objective function as a nested function.
- Calls `patternsearch`.

The following shows the code for the M-file.

```
function [x fval] =  runps(a,b,c,x0)
[x, fval] = patternsearch(@nestedfun,x0);
% Nested function that computes the objective function
    function y = nestedfun(x)
        y = (a - b*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + ...
         (-c + c*x(2)^2)*x(2)^2;
    end
end
```

Note that the objective function is computed in the nested function `nestedfun`, which has access to the variables `a`, `b`, and `c`. To run the optimization, enter

```
[x fval] = runps(a,b,c,x0)
```

This returns

```
Optimization terminated: current mesh size 9.5367e-007 is less
than 'TolMesh'.

x =

   -0.0898    0.7127


fval =

   -1.0316
```

6

Function Reference

Functions — Categorical List (p. 6-2)	Lists the functions in the toolbox by category.
Genetic Algorithm Options (p. 6-3)	Describes the options for the genetic algorithm.
Pattern Search Options (p. 6-21)	Describes the options for pattern search.
Functions — Alphabetical List (p. 6-37)	Lists the functions in the toolbox alphabetically.

Functions — Categorical List

The Genetic Algorithm and Direct Search Toolbox provides two categories of functions:

- Genetic algorithm
- Direct search

Genetic Algorithm

Function	Description
`ga`	Find the minimum of a function using the genetic algorithm
`gaoptimget`	Get values of a genetic algorithm `options` structure
`gaoptimset`	Create a genetic algorithm `options` structure
`gatool`	Open the Genetic Algorithm Tool

Direct Search

Function	Description
`patternsearch`	Find the minimum of a function using a pattern search
`psoptimget`	Get values of a pattern search `options` structure
`psoptimset`	Create a pattern search `options` structure
`psearchtool`	Open the Pattern Search Tool

Genetic Algorithm Options

This section describes the options for the genetic algorithm. There are two ways to specify options, depending on whether you are using the Genetic Algorithm Tool or calling the function `ga` at the command line:

- If you are using the Genetic Algorithm Tool (`gatool`), you specify the options by selecting an option from a drop-down list or by entering the value of the option in a text field. See "Setting Options in the Genetic Algorithm Tool" on page 4-12.
- If you are calling `ga` from the command line, you specify the options by creating an `options` structure using the function `gaoptimset`, as follows:

  ```
  options = gaoptimset('Param1', value1, 'Param2', value2, ...);
  ```

  See "Setting Options for ga at the Command Line" on page 4-22 for examples.

In this section, each option is listed in two ways:

- By its label, as it appears in the Genetic Algorithm Tool
- By its field name in the `options` structure

For example:

- **Population type** refers to the label of the option in the Genetic Algorithm Tool.
- `PopulationType` refers to the corresponding field of the `options` structure.

The genetic algorithm options are divided into the following categories:

- "Plot Options" on page 6-4
- "Population Options" on page 6-6
- "Fitness Scaling Options" on page 6-8
- "Selection Options" on page 6-9
- "Reproduction Options" on page 6-11
- "Mutation Options" on page 6-11
- "Crossover Options" on page 6-14
- "Migration Options" on page 6-16

- "Hybrid Function Option" on page 6-17
- "Stopping Criteria Options" on page 6-18
- "Output Function Options" on page 6-18
- "Display to Command Window Options" on page 6-19
- "Vectorize Option" on page 6-20

Plot Options

Plot options enable you to plot data from the genetic algorithm while it is running. When you select plot functions and run the genetic algorithm, a plot window displays the plots on separate axes. You can stop the algorithm at any time by clicking the **Stop** button on the plot window.

Plot interval (`PlotInterval`) specifies the number of generations between consecutive calls to the plot function.

You can select any of the following plot functions in the **Plots** pane:

- **Best fitness** (`@gaplotbestf`) plots the best function value versus generation.
- **Expectation** (`@gaplotexpectation`) plots the expected number of children versus the raw scores at each generation.
- **Score diversity** (`@gaplotscorediversity`) plots a histogram of the scores at each generation.
- **Stopping** (`@plotstopping`) plots stopping criteria levels.
- **Best individual** (`@gaplotbestindiv`) plots the vector entries of the individual with the best fitness function value in each generation.
- **Genealogy** (`@gaplotgenealogy`) plots the genealogy of individuals. Lines from one generation to the next are color-coded as follows:
  - Red lines indicate mutation children.
  - Blue lines indicate crossover children.
  - Black lines indicate elite individuals.
- **Scores** (`@gaplotscores`) plots the scores of the individuals at each generation.
- **Distance** (`@gaplotdistance`) plots the average distance between individuals at each generation.
- **Range** (`@gaplotrange`) plots the minimum, maximum, and mean fitness function values in each generation.

- **Selection** (`@gaplotselection`) plots a histogram of the parents.
- **Custom function** enables you to use plot functions of your own. To specify the plot function if you are using the Genetic Algorithm Tool,
  - Select **Custom function**.
  - Enter `@myfun` in the text box, where `myfun` is the name of your function.

  See "Structure of the Plot Functions" on page 6-5.

To display a plot when calling `ga` from the command line, set the `PlotFcns` field of `options` to be a function handle to the plot function. For example, to display the best fitness plot, set `options` as follows.

```
options = gaoptimset('PlotFcns', @gaplotbestf);
```

To display multiple plots, use the syntax

```
options =gaoptimset('PlotFcns', {@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are command-line names of plot functions.

Structure of the Plot Functions

The first line of a plot function has the form

```
function state = plotfun(options, state, flag)
```

The input arguments to the function are

- `options` — Structure containing all the current options settings
- `state` — Structure containing information about the current generation. "The State Structure" on page 6-5 describes the fields of `state`.
- `flag` — String that tells what stage the algorithm is currently in

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

The State Structure

The state structure, which is an input argument to plot, mutation, and output functions, contains the following fields:

- `Population` — Population in the current generation
- `Score` — Scores of the current population

- `Generation` — Current generation number
- `StartTime` — Time when GA started
- `StopFlag` — String containing the reason for stopping
- `Selection` — Indices of individuals selected for elite, crossover and mutation
- `Expectation` — Expectation for selection of individuals
- `Best` — Vector containing the best score in each generation
- `LastImprovement` — Generation at which the last improvement in fitness value occurred
- `LastImprovementTime` — Time at which last improvement occurred

Population Options

Population options enable you to specify the parameters of the population that the genetic algorithm uses.

Population type (`PopulationType`) specifies the data type of the input to the fitness function. You can set **Population type** to be one of the following:

- `Double Vector` (`'doubleVector'`) — Use this option if the individuals in the population have type `double`. This is the default.
- `Bit string` (`'bitstring'`) — Use this option if the individuals in the population are bit strings.
- `Custom` (`'custom'`) — Use this option to create a population whose data type is neither of the preceding.

  If you use a custom population type, you must write your own creation, mutation, and crossover functions that accept inputs of that population type, and specify these functions in the following fields, respectively:

  - **Creation function** (`CreationFcn`)
  - **Mutation function** (`MutationFcn`)
  - **Crossover function** (`CrossoverFcn`)

Population size (`PopulationSize`) specifies how many individuals there are in each generation. With a large population size, the genetic algorithm searches the solution space more thoroughly, thereby reducing the chance that the algorithm will return a local minimum that is not a global minimum. However, a large population size also causes the algorithm to run more slowly.

If you set **Population size** to a vector, the genetic algorithm creates multiple subpopulations, the number of which is the length of the vector. The size of each subpopulation is the corresponding entry of the vector.

Creation function (`CreationFcn`) specifies the function that creates the initial population for `ga`. You can choose from the following functions:

- `Uniform` (`@gacreationuniform`) creates a random initial population with a uniform distribution. This is the default.
- `Custom` enables you to write your own creation function, which must generate data of the type that you specify in **Population type**. To specify the creation function if you are using the Genetic Algorithm Tool,
  - Set **Creation function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('CreationFcn', @myfun);
  ```

  Your creation function must have the following calling syntax.

  ```
  function Population = myfun(GenomeLength, FitnessFcn, options)
  ```

  The input arguments to the function are
  - `Genomelength` — Number of independent variables for the fitness function
  - `FitnessFcn` — Fitness function
  - `options` — Options structure

  The function returns `Population`, the initial population for the genetic algorithm.

  "Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

Initial population (`InitialPopulation`) specifies an initial population for the genetic algorithm. The default value is `[]`, in which case `ga` uses the **Creation function** to create an initial population. If you enter a nonempty array in the **Initial population** field, the array must have **Population size** rows and **Number of variables** columns. In this case, the genetic algorithm does not call the **Creation function**.

Initial scores (`InitialScores`) specifies initial scores for the initial population.

Initial range (`PopInitRange`) specifies the range of the vectors in the initial population that is generated by the creation function. You can set **Initial range** to be a matrix with two rows and **Number of variables** columns, each column of which has the form `[lb; ub]`, where `lb` is the lower bound and `ub` is the upper bound for the entries in that coordinate. If you specify **Initial range** to be a 2-by-1 vector, each entry is expanded to a constant row of length **Number of variables**.

See "Example —Setting the Initial Range" on page 4-31 for an example.

Fitness Scaling Options

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. You can specify options for fitness scaling in the **Fitness scaling** pane.

Scaling function (`FitnessScalingFcn`) specifies the function that performs the scaling. The options are

- `Rank` (`@fitscalingrank`) — The default fitness scaling function, `Rank`, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores. The rank of the most fit individual is 1, the next most fit is 2, and so on. Rank fitness scaling removes the effect of the spread of the raw scores.
- `Proportional` (`@fitscalingprop`) — Proportional scaling makes the scaled value of an individual proportional to its raw fitness score.
- `Top` (`@fitscalingtop`) — Top scaling scales the top individuals equally. Selecting `Top` displays an additional field, **Quantity**, which specifies the number of individuals that are assigned positive scaled values. **Quantity** can be an integer between 1 and the population size or a fraction between 0 and 1 specifying a fraction of the population size. The default value is `0.4`. Each of the individuals that produce offspring is assigned an equal scaled value, while the rest are assigned the value 0. The scaled values have the form [0 1/n 1/n 0 0 1/n 0 0 1/n ...].

  To change the default value for **Quantity** at the command line, use the following syntax:

  ```
  options = gaoptimset('FitnessScalingFcn', {@fitscalingtop,
  quantity})
  ```

  where `quantity` is the value of **Quantity**.

- `Shift linear` (`@fitscalingshiftlinear`) — Shift linear scaling scales the raw scores so that the expectation of the fittest individual is equal to a constant multiplied by the average score. You specify the constant in the **Max survival rate** field, which is displayed when you select `Shift linear`. The default value is `2`.

  To change the default value of **Max survival rate** at the command line, use the following syntax:

  ```
  options = gaoptimset('FitnessScalingFcn',
  {@fitscalingshiftlinear, rate})
  ```

  where `rate` is the value of **Max survival rate**.
- `Custom` enables you to write your own scaling function. To specify the scaling function using the Genetic Algorithm Tool,
  - Set **Scaling function** to `Custom`
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga` at the command line, set

  ```
  options = gaoptimset('FitnessScalingFcn', @myfun);
  ```

  Your scaling function must have the following calling syntax.

  ```
  function expection = myfun(scores, nParents)
  ```

  The input arguments to the function are
  - `scores` — A vector of scalars, one for each member of the population
  - `nParents` — The number of parents needed from this population

  The function returns `expectation`, a row vector of scalars of the same length as `scores`, giving the scaled values of each member of the population. The sum of the entries of `expectation` must equal `nParents`.

  "Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

See "Fitness Scaling" on page 4-35 for more information.

Selection Options

Selection options specify how the genetic algorithm chooses parents for the next generation. You can specify the function the algorithm uses in the

Selection function (`SelectionFcn`) field in the **Selection** options pane. The options are

- `Stochastic uniform` (`@selectionstochunif`) — The default selection function, `Stochastic uniform`, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on. The first step is a uniform random number less than the step size.
- `Remainder` (`@selectionremainder`) — Remainder selection assigns parents deterministically from the integer part of each individual's scaled value and then uses roulette selection on the remaining fractional part. For example, if the scaled value of an individual is 2.3, that individual is listed twice as a parent because the integer part is 2. After parents have been assigned according to the integer parts of the scaled values, the rest of the parents are chosen stochastically. The probability that a parent is chosen in this step is proportional to the fractional part of its scaled value.
- `Uniform` (`@selectionuniform`) — Uniform selection chooses parents using the expectations and number of parents. Uniform selection is useful for debugging and testing, but is not a very effective search strategy.
- `Roulette` (`@selectionroulette`) — Roulette selection chooses parents by simulating a roulette wheel, in which the area of the section of the wheel corresponding to an individual is proportional to the individual's expectation. The algorithm uses a random number to select one of the sections with a probability equal to its area.
- `Tournament` (`@selectiontournament`) — Tournament selection chooses each parent by choosing **Tournament size** players at random and then choosing the best individual out of that set to be a parent. **Tournament size** must be at least 2. The default value of **Tournament size** is `4`.

  To change the default value of **Tournament size** at the command line, use the syntax

  ```
  options = gaoptimset('SelectionFcn', {@selecttournament, size})
  ```

  where `size` is the value of **Tournament size**.
- `Custom` enables you to write your own selection function. To specify the selection function using the Genetic Algorithm Tool,
  - Set **Selection function** to `Custom`

- Set **Function name** to `@myfun`, where `myfun` is the name of your function.

If you are using `ga` at the command line, set

```
options = gaoptimset('SelectionFcn', @myfun);
```

Your selection function must have the following calling syntax:

```
function parents = myfun(expectation, nParents, options)
```

The input arguments to the function are

- `expectation` — Expected number of children for each member of the population
- `nParents` — Number of parents to select
- `options` — Genetic algorithm `options` structure

The function returns `parents`, a row vector of length `nParents` containing the indices of the parents that you select.

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

See "Selection" on page 4-39 for more information.

Reproduction Options

Reproduction options specify how the genetic algorithm creates children for the next generation.

Elite count (`EliteCount`) specifies the number of individuals that are guaranteed to survive to the next generation. Set **Elite count** to be a positive integer less than or equal to the population size. The default value is `2`.

Crossover fraction (`CrossoverFraction`) specifies the fraction of the next generation, other than elite children, that are produced by crossover. Set **Crossover fraction** to be a fraction between `0` and `1`, either by entering the fraction in the text box or moving the slider. The default value is `0.8`.

See "Setting the Crossover Fraction" on page 4-43 for an example.

Mutation Options

Mutation options specify how the genetic algorithm makes small random changes in the individuals in the population to create mutation children. Mutation provides genetic diversity and enable the genetic algorithm to search

a broader space. You can specify the mutation function in the **Mutation function** (`MutationFcn`) field in the **Mutation** options pane. You can choose from the following functions:

- `Gaussian` (`mutationgaussian`) — The default mutation function, `Gaussian`, adds a random number taken from a Gaussian distribution with mean 0 to each entry of the parent vector. The variance of this distribution is determined by the parameters **Scale** and **Shrink**, which are displayed when you select `Gaussian`, and by the **Initial range** setting in the **Population** options.
  - The **Scale** parameter determines the variance at the first generation. If you set **Initial range** to be a 2-by-1 vector `v`, the initial variance is the same at all coordinates of the parent vector, and is given by **Scale***`(v(2) - v(1))`.

    If you set **Initial range** to be a vector `v` with two rows and **Number of variables** columns, the initial variance at coordinate `i` of the parent vector is given by **Scale***`(v(i,2) - v(i,1))`.
  - The **Shrink** parameter controls how the variance shrinks as generations go by. If you set **Initial range** to be a 2-by-1 vector, the variance at the kth generation, var_k, is the same at all coordinates of the parent vector, and is given by the recursive formula

    $$var_k = var_{k-1}\left(1 - \textbf{Shrink} \cdot \frac{k}{\textbf{Generations}}\right)$$

    If you set **Initial range** to be a vector with two rows and **Number of variables** columns, the variance at coordinate i of the parent vector at the kth generation, $var_{i,k}$, is given by the recursive formula

    $$var_{i,k} = var_{i,k-1}\left(1 - \textbf{Shrink} \cdot \frac{k}{\textbf{Generations}}\right)$$

    If you set **Shrink** to `1`, the algorithm shrinks the variance in each coordinate linearly until it reaches 0 at the last generation is reached. A negative value of **Shrink** causes the variance to grow.

  The default values of **Scale** and **Shrink** are `0.5` and `0.75`, respectively. To change these default values at the command line, use the syntax

```
options = gaoptimset('MutationFcn', ...
{@mutationgaussian, scale, shrink})
```

where `scale` and `shrink` are the values of **Scale** and **Shrink**, respectively.

- `Uniform` (`mutationuniform`) — Uniform mutation is a two-step process. First, the algorithm selects a fraction of the vector entries of an individual for mutation, where each entry has a probability **Rate** of being mutated. The default value of **Rate** is `0.01`. In the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry.

  To change the default value of **Rate** at the command line, use the syntax

  ```
  options = gaoptimset('MutationFcn', {@mutationuniform, rate})
  ```

  where `rate` is the value of **Rate**.

- `Custom` enables you to write your own mutation function. To specify the mutation function using the Genetic Algorithm Tool,

  - Set **Mutation function** to `Custom`
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('MutationFcn', @myfun);
  ```

  Your mutation function must have this calling syntax:

  ```
  function mutationChildren = myfun(parents, options, nvars,
  FitnessFcn, state, thisScore, thisPopulation)
  ```

  The arguments to the function are

  - `parents` — Row vector of parents chosen by the selection function
  - `options` — Options structure
  - `nvars` — Number of variables
  - `FitnessFcn` — Fitness function
  - `state` — Structure containing information about the current generation. "The State Structure" on page 6-5 describes the fields of `state`.
  - `thisScore` — Vector of scores of the current population

- `thisPopulation` — Matrix of individuals in the current population

The function returns `mutationChildren` — the mutated offspring — as a matrix whose rows correspond to the children. The number of columns of the matrix is **Number of variables**.

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

Crossover Options

Crossover options specify how the genetic algorithm combines two individuals, or parents, to form a crossover child for the next generation.

Crossover function (`CrossoverFcn`) specifies the function that performs the crossover. You can choose from the following functions:

- `Scattered` (`@crossoverscattered`), the default crossover function, creates a random binary vector and selects the genes where the vector is a 1 from the first parent, and the genes where the vector is a 0 from the second parent, and combines the genes to form the child. For example, if `p1` and `p2` are the parents

  ```
  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]
  ```

  and the binary vector is [1 1 0 0 1 0 0 0], the function returns the following child:

  ```
  child1 = [a b 3 4 e 6 7 8]
  ```

- `Single point` (@`crossoversinglepoint`) chooses a random integer n between 1 and **Number of variables** and then
  - Selects vector entries numbered less than or equal to n from the first parent.
  - Selects vector entries numbered greater than n from the second parent.
  - Concatenates these entries to form a child vector.

    For example, if `p1` and `p2` are the parents

  ```
  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]
  ```

  and the crossover point is 3, the function returns the following child.

  ```
  child = [a b c 4 5 6 7 8]
  ```

- `Two point` (`@crossovertwopoint`) selects two random integers m and n between 1 and **Number of variables**. The function selects
  - Vector entries numbered less than or equal to m from the first parent
  - Vector entries numbered from m+1 to n, inclusive, from the second parent
  - Vector entries numbered greater than n from the first parent.

  The algorithm then concatenates these genes to form a single gene. For example, if `p1` and `p2` are the parents

  ```
  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]
  ```

  and the crossover points are 3 and 6, the function returns the following child.

  ```
  child = [a b c 4 5 6 g h]
  ```

- `Intermediate` (`@crossoverintermediate`) creates children by taking a weighted average of the parents. You can specify the weights by a single parameter, **Ratio**, which can be a scalar or a row vector of length **Number of variables**. The default is a vector of all 1's. The function creates the child from `parent1` and `parent2` using the following formula.

  ```
  child = parent1 + rand * Ratio * ( parent2 - parent1)
  ```

  If all the entries of **Ratio** lie in the range [0, 1], the children produced are within the hypercube defined by placing the parents at opposite vertices. If **Ratio** is not in that range, the children might lie outside the hypercube. If **Ratio** is a scalar, then all the children lie on the line between the parents.

  To change the default value of **Ratio** at the command line, use the syntax

  ```
  options = gaoptimset('CrossoverFcn', ...
  {@crossoverintermediate, ratio});
  ```

  where `ratio` is the value of **Ratio**.

- `Heuristic` (`@crossoverheuristic`) returns a child that lies on the line containing the two parents, a small distance away from the parent with the better fitness value in the direction away from the parent with the worse fitness value. You can specify how far the child is from the better parent by the parameter **Ratio**, which appears when you select `Heuristic`. The default value of **Ratio** is 1.2. If `parent1` and `parent2` are the parents, and `parent1` has the better fitness value, the function returns the child

```
child = parent2 + R * (parent1 - parent2);
```

To change the default value of **Ratio** at the command line, use the syntax

```
options=gaoptimset('CrossoverFcn',{@crossoverheuristic,ratio});
```

where `ratio` is the value of **Ratio**.

- `Custom` enables you to write your own crossover function. To specify the crossover function using the Genetic Algorithm Tool,
  - Set **Crossover function** to `Custom`.
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `ga`, set

  ```
  options = gaoptimset('CrossoverFcn', @myfun);
  ```

  Your selection function must have the following calling syntax.

  ```
  xoverKids = myfun(parents, options, nvars, FitnessFcn,
  unused,thisPopulation)
  ```

  The arguments to the function are
  - `parents` — Row vector of parents chosen by the selection function
  - `options` — `options` structure
  - `nvars` — Number of variables
  - `FitnessFcn` — Fitness function
  - `unused` — Place holder that is not used
  - `thisPopulation` — Matrix representing the current population. The number of rows of the matrix is **Population size** and the number of columns is **Number of variables**.

  The function returns `xoverKids` — the crossover offspring — as a matrix whose rows correspond to the children. The number of columns of the matrix is **Number of variables**.

  "Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

Migration Options

Migration options specify how individuals move between subpopulations. Migration occurs if you set **Population size** to be a vector of length greater than 1. When migration occurs, the best individuals from one subpopulation

replace the worst individuals in another subpopulation. Individuals that migrate from one subpopulation to another are copied. They are not removed from the source subpopulation.

You can control how migration occurs by the following three fields in the **Migration** options pane:

- **Direction** (`MigrationDirection`) — Migration can take place in one or both directions.
  - If you set **Direction** to `Forward` (`'forward'`), migration takes place toward the last subpopulation. That is the nth subpopulation migrates into the (n+1)th subpopulation.
  - If you set **Direction** to `Both` (`'both'`), the nth subpopulation migrates into both the (n-1)th and the (n+1)th subpopulation.

  Migration wraps at the ends of the subpopulations. That is, the last subpopulation migrates into the first, and the first may migrate into the last. To prevent wrapping, specify a subpopulation of size 0 by adding an entry of `0` at the end of the population size vector that you enter in **Population size**.
- **Interval** (`MigrationInterval`) — Specifies how many generation pass between migrations. For example, if you set **Interval** to `20`, migration takes place every 20 generations.
- **Fraction** (`MigrationFraction`) — Specifies how many individuals move between subpopulations. **Fraction** specifies the fraction of the smaller of the two subpopulations that moves. For example, if individuals migrate from a subpopulation of 50 individuals into a subpopulation of 100 individuals and you set **Fraction** to `0.1`, the number of individuals that migrate is 0.1 * 50 = 5.

Hybrid Function Option

A hybrid function is another minimization function that runs after the genetic algorithm terminates. You can specify a hybrid function in **Hybrid function** (`HybridFcn`) options. The choices are

- `[]` — No hybrid function
- `fminsearch` (`@fminsearch`) — Uses the MATLAB function `fminsearch`
- `patternsearch` (`@patternsearch`) — Uses a pattern search
- `fminunc` (`@fminunc`) — Uses the Optimization Toolbox function `fminunc`

See “Using a Hybrid Function” on page 4-52 for an example.

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- **Generations** (`Generations`) — Specifies the maximum number of iterations the genetic algorithm will perform. The default is `100`.
- **Time limit** (`TimeLimit`) — Specifies the maximum time in seconds the genetic algorithm runs before stopping.
- **Fitness limit** (`FitnessLimit`) — The algorithm stops if the best fitness value is less than or equal to the value of **Fitness limit**.
- **Stall generations** (`StallGenLimit`) — The algorithm stops if there is no improvement in the best fitness value for the number of generations specified by **Stall generations**.
- **Stall time** (`StallTimeLimit`) — The algorithm stops if there is no improvement in the best fitness value for an interval of time in seconds specified by **Stall time**.

See “Setting the Maximum Number of Generations” on page 4-54 for an example.

Output Function Options

Output functions return output from the genetic algorithm to the command line at each generation.

History to new window (`@gaoutputgen`) displays the history of points computed by the algorithm in a new window at each multiple of **Interval** iterations.

Custom enables you to write your own output function. To specify the output function using the Genetic Algorithm Tool,

- Select **Custom function**.
- Enter `@myfun` in the text box, where `myfun` is the name of your function.

If you are using `ga`, set

```
options = gaoptimset('OutputFcn', @myfun);
```

To see a template that you can use to write your own output functions, enter

```
edit gaoutputfcntemplate
```

at the MATLAB command line.

The following section describe the structure of the output function.

Structure of the Output Function

The output function has the following calling syntax.

```
[state, options,optchanged] = myfun(options,state,flag,interval)
```

The function has the following input arguments:

- `options` — Options structure
- `state` — Structure containing information about the current generation. "The State Structure" on page 6-5 describes the fields of `state`.
- `flag` — String indicating the current status of the algorithm as follows:
  - `'init'` — Initial stage
  - `'iter'` — Algorithm running
  - `'done'` — Algorithm terminated
- `interval` — Optional interval argument

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

The output function returns the following arguments to `ga`:

- `state` — Structure containing information about the current generation
- `options` — Options structure modified by the output function. This argument is optional.
- `optchanged` — Flag indicating changes to `options`

Display to Command Window Options

Level of display (`'Display'`) specifies how much information is displayed at the command line while the genetic algorithm is running. The available options are

- `Off` (`'off'`) — Only the final answer is displayed.

- `Iterative` (`'iter'`) — Information is displayed at each iteration.
- `Diagnose` (`'diagnose'`) — Information is displayed at each iteration. In addition, options that are changed from the defaults are listed.
- `Final` (`'final'`) — The outcome of the genetic algorithm (successful or unsuccessful), the reason for stopping, and the final point.

Both `Iterative` and `Diagnose` display the following information:

- `Generation` — Generation number
- `f-count` — Cumulative number of fitness function evaluations
- `Best f(x)` — Best fitness function value
- `Mean f(x)` — Mean fitness function value
- `Stall Generations` — Number of generations since the last improvement of the fitness function

The default value of **Level of display** is

- `Off` in the Genetic Algorithm Tool
- `'final'` in an options structure created using `gaoptimset`

Vectorize Option

The vectorize option specifies whether the computation of the fitness function is vectorized. When you set **Fitness function is vectorized** (`Vectorized`) to `Off`, the genetic algorithm computes the fitness function values of the new generation in a loop. When you set **Fitness function is vectorized** to `On`, the algorithm computes the fitness function values of a new generation with one call to the fitness function, which is faster than computing the values in a loop. However, to use this option, your fitness function must be able to accept input matrices with an arbitrary number of rows.

See “Vectorizing the Fitness Function” on page 4-56 for an example.

Pattern Search Options

This section describes the options for pattern search. There are two ways to specify the options, depending on whether you are using the Pattern Search Tool or calling the function `patternsearch` at the command line:

- If you are using the Pattern Search Tool (`psearchtool`), you specify the options by selecting an option from a drop-down list or by entering the value of the option in the text field, as described in "Setting Options in the Pattern Search Tool" on page 5-10.
- If you are calling `patternsearch` from the command line, you specify the options by creating an `options` structure using the function `psoptimset`, as follows:

  ```
  options = psoptimset('Param1', value1, 'Param2', value2, ...);
  ```

  See "Setting Options for patternsearch at the Command Line" on page 5-16 for examples.

In this section, each option is listed in two ways:

- By its label, as it appears in the Pattern Search Tool
- By its field name in the `options` structure

For example:

- **Poll method** refers to the label of the option in the Pattern Search Tool.
- `PollMethod` refers to the corresponding field of the `options` structure.

The options are divided into the following categories:

- "Plot Options" on page 6-22
- "Poll Options" on page 6-24
- "Search Options" on page 6-26
- "Mesh Options" on page 6-30
- "Cache Options" on page 6-31
- "Stopping Criteria" on page 6-31
- "Output Function Options" on page 6-32

- "Display to Command Window Options" on page 6-34
- "Vectorize Option" on page 6-34

Plot Options

Plot options enable you to plot data from the pattern search while it is running. When you select plot functions and run the pattern search, a plot window displays the plots on separate axes. You can stop the algorithm at any time by clicking the **Stop** button on the plot window.

Plot interval (`PlotInterval`) specifies the number of iterations between consecutive calls to the plot function.

You can select any of the following plots in the **Plots** pane.

- **Best function value** (`@psplotbestf`) plots the best objective function value.
- **Function count** (`@psplotfuncount`) plots the number of function evaluations.
- **Mesh size** (`@psplotmeshsize`) plots the mesh size.
- **Best point** (`@psplotbestx`) plots the current best point.
- **Custom** enables you to use your own plot function. To specify the plot function using the Pattern Search Tool,
  - Select **Custom function**.
  - Enter `@myfun` in the text box, where `myfun` is the name of your function.

  "Structure of the Plot Functions" on page 6-23 describes the structure of a plot function.

To display a plot when calling `patternsearch` from the command line, set the `PlotFcns` field of `options` to be a function handle to the plot function. For example, to display the best function value, set `options` as follows.

```
options = psoptimset('PlotFcns', @psplotbestf);
```

To display multiple plots, use the syntax

```
options = psoptimset('PlotFcns', {@plotfun1, @plotfun2, ...});
```

where `@plotfun1`, `@plotfun2`, and so on are command-line names of plot functions (listed in parentheses in the preceding list).

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(optimvalues, flag)
```

The input arguments to the function are

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point
  - `iteration` — Iteration number
  - `fval` — Objective function value
  - `meshsize` — Current mesh size
  - `funccount` — Number of function evaluations
  - `method` — Method used in last iteration
  - `TolFun` — Tolerance on function value in last iteration
  - `TolX` — Tolerance on `x` value in last iteration
- `flag` — Current state in which the plot function is called. The possible values for `flag` are
  - `init` — initialization state
  - `iter` — iteration state
  - `done` — final state

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the function.

The output argument `stop` provides a way to stop the algorithm at the current iteration. `stop` can have the following values:

- `false` — The algorithm continues to the next iteration.
- `true` — The algorithm terminates at the current iteration.

Poll Options

Poll options control how the pattern search polls the mesh points at each iteration.

Poll method (`PollMethod`) specifies the pattern the algorithm uses to create the mesh. There are two patterns:

- The default pattern, `Positive basis 2N`, consists of the following 2N vectors, where N is the number of independent variables for the objective function.

  [100…0]
  [010…0]
  …
  [000…1]
  [–1 00…0]
  [0 –10…0]
  …
  [0 00…–1]

  For example, if objective function has three independent variables, the pattern consists of the following six vectors.

  $$\begin{bmatrix}1 & 0 & 0\end{bmatrix}$$

  $$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$$

  $$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$$

  $$\begin{bmatrix}-1 & 0 & 0\end{bmatrix}$$

  $$\begin{bmatrix}0 & -1 & 0\end{bmatrix}$$

  $$\begin{bmatrix}0 & 0 & -1\end{bmatrix}$$

- The `Positive basis NP1` pattern consisting of the following N + 1 vectors.

  [100…0]
  [010…0]
  …
  [000…1]
  [–1 –1–1…–1]

  For example, if objective function has three independent variables, the pattern consists of the following four vectors.

  $$\begin{bmatrix}1 & 0 & 0\end{bmatrix}$$
  $$\begin{bmatrix}0 & 1 & 0\end{bmatrix}$$
  $$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$$
  $$\begin{bmatrix}-1 & -1 & -1\end{bmatrix}$$

Complete poll (`CompletePoll`) specifies whether all the points in the current mesh must be polled at each iteration. **Complete Poll** can have the values `On` or `Off`.

- If you set **Complete poll** to `On`, the algorithm polls all the points in the mesh at each iteration and chooses the point with the smallest objective function value as the current point at the next iteration.
- If you set **Complete poll** to `Off`, the default value, the algorithm stops the poll as soon as it finds a point whose objective function value is less than that of the current point. The algorithm then sets that point as the current point at the next iteration.

Polling order (`PollingOrder`) specifies the order in which the algorithm searches the points in the current mesh. The options are

- `Random` — The polling order is random.
- `Success` — The first search direction at each iteration is the direction in which the algorithm found the best point at the previous iteration. After the first point, the algorithm polls the mesh points in the same order as **Consecutive**.

- `Consecutive` — The algorithm polls the mesh points in *consecutive* order, that is, the order of the pattern vectors as described in Poll method.

See "Poll Options" on page 6-24 for more information.

Search Options

Search options specify an optional search that the algorithm can perform at each iteration prior to the polling. If the search returns a point that improves the objective function, the algorithm uses that point at the next iteration and omits the polling.

Complete search (`CompleteSearch`) only applies when you set **Search method** to `Positive basis Np1`, `Positive basis 2N`, or `Latin hypercube`. **Complete search** can have the values `On` or `Off`.

For `Positive basis Np1` or `Positive basis 2N`, **Complete search** has the same meaning as the poll option **Complete poll**.

Search method (`SearchMethod`) specifies the method of the search. The options are

- `None` (`[]`) specifies no search (the default).
- `Positive basis Np1` (`'PositiveBasisNp1'`) specifies a pattern search using the `Positive Basis Np1` option for **Poll method**.
- `Positive basis 2N` (`'PositiveBasis2N'`) specifies a pattern search using the `Positive Basis 2N` option for **Poll method**.
- `Genetic Algorithm` (`@searchga`) specifies a search using the genetic algorithm. If you select `Genetic Algorithm`, two other options appear:
  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the genetic algorithm search is performed.
  - **Options** — Options structure for the genetic algorithm, which you can set using `gaoptimset`

  To change the default values of **Iteration limit** and **Options** at the command line, use the syntax

  ```
  options=psoptimset('SearchMethod', {@searchga,iterlim,optionsGA}
  ```

  where `iterlim` is the value of **Iteration limit** and `optionsGA` is the genetic algorithm options structure.

- `Latin hypercube` (`@searchlhs`) specifies a Latin hypercube search. The way the search is performed depends on the setting for **Complete search**:
  - If you set **Complete search** to `On`, the algorithm polls all the points that are randomly generated at each iteration by the latin hypercube search and chooses the one with the smallest objective function value.
  - If you set **Complete search** to `Off`, the algorithm stops the poll as soon as it finds one of the randomly generated points whose objective function value is less than that of the current point, and chooses that point for the next iteration.

  If you select `Latin hypercube`, two other options appear:
  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the Latin hypercube search is performed.
  - **Design level** — A positive integer specifying the design level. The number of points searched equals the **Design level** multiplied by the number of independent variables for the objective function.

  To change the default values of **Iteration limit** and **Design level** at the command line, use the syntax

  ```
  options=psoptimset('SearchMethod', {@searchlhs,iterlim,level}
  ```

  where `iterlim` is the value of **Iteration limit** and `level` is the value of **Design level**.

- `Nelder-Mead` (`@searchneldermead`) specifies a search using `fminsearch`, which uses the Nelder-Mead algorithm. If you select `Nelder-Mead`, two other options appear:
  - **Iteration limit** — Positive integer specifying the number of iterations of the pattern search for which the Nelder-Mead search is performed
  - **Options** — Options structure for the function `fminsearch`, which you can create using the function `optimset`.

  To change the default values of **Iteration limit** and **Options** at the command line, use the syntax

  ```
  options=psoptimset('SearchMethod', {@searchga,iterlim,optionsNM}
  ```

  where `iterlim` is the value of **Iteration limit** and `optionsNM` is the options structure.

- `Custom` enables you to write your own search function. To specify the search function using the Pattern Search Tool,
  - Set **Search function** to `Custom`
  - Set **Function name** to `@myfun`, where `myfun` is the name of your function.

  If you are using `patternsearch`, set

  ```
  options = psoptimset('SearchMethod', @myfun);
  ```

  To see a template that you can use to write your own search function, enter

  ```
  edit searchfcntemplate
  ```

  The following section describes the structure of the search function.

Structure of the Search Function

Your search function must have the following calling syntax.

```
function [successSearch,nextIterate,FunEval] =
searchfcntemplate(fun,x0,iterate,tol,A,L,U, ...
    funeval,maxfun,searchoptions,objfcnarg)
```

The search function has the following input arguments:

- `fun` — Objective function
- `xin` — Initial point
- `iterate` — Current point in the iteration. Iterate is a structure that contains the current point and the function value.
- `tol` — Tolerance that determines whether the constraints are active or not
- `A`, `L`, `U` — Defines the feasible region in case of linear or bound constraints as `L <= A*x <= U`.
- `funeval` — Counter for number of function evaluations. `Funeval` is always less than `maxfun`, which is maximum number of function evaluations.
- `maxfun` — Maximum limit on number of function evaluations.
- `searchoptions` — Structure that enables you to set search options. The structure contains the following fields:
  - `completesearch` — If `'off'`, the search can be terminated as soon as a better point is found; that is, no sufficient decrease condition is imposed. The default is `'on'`. See `psoptimset` for a description of `completesearch`.

  - `meshsize` — Current mesh size used in search step
  - `iteration` — Current iteration number
  - `scale` — Scale factor used to scale the design points
  - `indineqcstr` — Indices of inequality constraints
  - `indeqcstr` — Indices of equality constraints
  - `problemtype` — Flag passed to the search routines, indicating whether the problem is `'unconstrained'`, `'boundconstraints'`, or `'linearconstraints'`.
  - `notvectorized` — A flag indicating `fun` is not evaluated as vectorized
  - `cache` — A flag for using cache. If `'off'`, no cache is used.
  - `cachetol` — Tolerance used in cache to determine whether two points are the same or not
  - `cachelimit` — Limit to the cache size
- `objfunarg` — Cell array of additional arguments for objective function.

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the search function.

The function has the following output arguments:

- `successsearch` — A Boolean identifier indicating whether the search is successful or not
- `nextiterate` — Successful iterate after polling is done. If poll is not successful, `nextiterate` is same as `iterate`.

Note If you set **Search method** to `Genetic algorithm` or `Nelder-Mead`, we recommend that you leave **Iteration limit** set to the default value `1`, as performing these searches more than once is not likely to improve results.

See "Using a Search Method" on page 5-25 for an example.

Mesh Options

Mesh options control the mesh that the pattern search uses. The following options are available.

Initial size (`InitialMeshSize`) specifies the size of the initial mesh, which is the length of the shortest vector from the initial point to a mesh point. **Initial size** should be a positive scalar. The default is `1.0`.

Max size (`MaxMeshSize`) specifies a maximum size for the mesh. When the maximum size is reached, the mesh size does not increase after a successful iteration. **Max size** must be a positive scalar. The default value is `Inf`.

Accelerator (`MeshAccelerator`) specifies whether the **Contraction factor** is multiplied by `0.5` after each unsuccessful iteration. **Accelerator** can have the values `On` or `Off`, the default.

Rotate (`MeshRotate`) specifies whether the mesh vectors are multiplied by -1 when the mesh size is less than a small value. **Rotate** is only applied when **Poll method** is set to `Positive basis Np1` and **Rotate** is set to `On`, the default.

Note Changing the setting of **Rotate** has no effect on the poll when **Poll method** is set to `Positive basis 2N`.

Scale (`ScaleMesh`) specifies whether the algorithm scales the mesh points by multiplying the pattern vectors by constants. **Scale** can have the values `Off` or `On` (the default).

Expansion factor (`MeshExpansion`) specifies the factor by which the mesh size is increased after a successful poll. The default value is `2.0`, which means that the size of the mesh is multiplied by `2.0` after a successful poll. **Expansion factor** must be a positive scalar.

Contraction factor(`MeshContraction`) specifies the factor by which the mesh size is decreased after an unsuccessful poll. The default value is `0.5`, which means that the size of the mesh is multiplied by `0.5` after an unsuccessful poll. **Contraction factor** must be a positive scalar.

See “Mesh Expansion and Contraction” on page 5-28 for more information.

Cache Options

The pattern search algorithm can keep a record of the points it has already polled, so that it does not have to poll the same point more than once. If the objective function requires a relatively long time to compute, the cache option can speed up the algorithm. The memory allocated for recording the points is called the cache. This option should only be used for deterministic objective functions, but not for stochastic ones.

Cache (`Cache`) specifies whether a cache is used. The options are `On` and `Off`, the default. When you set **Cache** to `On`, the algorithm does not evaluate the objective function at any mesh points that are within **Tolerance** of a point in the cache.

Tolerance (`CacheTol`) specifies how close a mesh point must be to a point in the cache for the algorithm to omit polling it. **Tolerance** must be a positive scalar. The default value is `eps`.

Size (`CacheSize`) specifies the size of the cache. **Size** must be a positive scalar. The default value is `1e4`.

See “Using Cache” on page 5-34 for more information.

Stopping Criteria

Stopping criteria determine what causes the pattern search algorithm to stop. Pattern search uses the following criteria:

Mesh tolerance (`TolMesh`) specifies the minimum tolerance for mesh size. The algorithm stops if the mesh size becomes smaller than **Mesh tolerance**. The default value is `1e-6`.

Max iteration (`MaxIter`) specifies the maximum number of iterations the algorithm performs. The algorithm stops if the number of iterations reaches **Max iteration**. You can select either

- **100*numberofvariables** — Maximum number of iterations is 100 times the number of independent variables (the default).
- **Specify** — A positive integer for the maximum number of iterations

Max function evaluations (`MaxFunEval`) specifies the maximum number of evaluations of the fitness function. The algorithm stops if the number of function evaluations reaches **Max function evaluations**. You can select either

- **2000*numberofvariables** — Maximum number of function evaluations is `2000` times the number of independent variables.
- **Specify** — A positive integer for the maximum number of function evaluations

Bind tolerance (`TolBind`) specifies the minimum tolerance for the distance from the current point to the boundary of the feasible region. `Bind tolerance` specifies when a linear constraint is active. It is not a stopping criterion. The default value is `1e-3`.

X tolerance (`TolX`) specifies the minimum distance between the current points at two consecutive iterations. The algorithm stops if the distance between two consecutive points is less than **X tolerance**. The default value is `1e-6`.

Function tolerance (`TolFun`) specifies the minimum tolerance for the objective function. The algorithm stops when the value of the objective function at the current point is less than **Function tolerance**. The default value is `1e-6`.

See “Setting Tolerances for the Solver” on page 5-36 for an example.

Output Function Options

Output functions are functions that the pattern search algorithm calls at each iteration. The following options are available:

- **History to new window** (`@psoutputhistory`) displays the history of points computed by the algorithm in the MATLAB Command Window at each multiple of **Interval** iterations.
- **Custom** enables you to write your own output function. To specify the output function using the Pattern Search Tool,
  - Select **Custom function**.
  - Enter `@myfun` in the text box, where `myfun` is the name of your function.

  If you are using `patternsearch`, set

  ```
  options = psoptimset('OutputFcn', @myfun);
  ```

  To see a template that you can use to write your own output function, enter

  ```
  edit psoutputfcntemplate
  ```

The following section describes the structure of the output function.

Structure of the Output Function

Your output function must have the following calling syntax:

```
[stop, options,optchanged] =
psoutputhistory(optimvalues,options,flag,interval)
```

The function has the following input arguments:

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:
  - `x` — Current point
  - `iteration` — Iteration number
  - `fval` — Objective function value
  - `meshsize` — Current mesh size
  - `funccount` — Number of function evaluations
  - `method` — Method used in last iteration
  - `TolFun` — Tolerance on function value in last iteration
  - `TolX` — Tolerance on `x` value in last iteration
- `options` — Options structure
- `flag` — Current state in which the output function is called. The possible values for `flag` are
  - `init` — initialization state
  - `iter` — iteration state
  - `done` — final state
- `interval` — Optional interval argument

"Parameterizing Functions Called by patternsearch or ga" on page 5-42 explains how to provide additional parameters to the output function.

The output function returns the following arguments to `ga`:

- `stop` — Provides a way to stop the algorithm at the current iteration. `stop` can have the following values:
  - `false` — The algorithm continues to the next iteration.
  - `true` — The algorithm terminates at the current iteration.

- `options` — Options structure
- `optchanged` — Flag indicating changes to `options`

Display to Command Window Options

Level of display (`'Display'`) specifies how much information is displayed at the command line while the pattern search is running. The available options are

- `Off` (`'off'`) — Only the final answer is displayed.
- `Iterative` (`'iter'`) — Information is displayed for each iteration.
- `Diagnose` (`'diagnose'`) — Information is displayed for each iteration. In addition, the options that have been changed from the defaults are listed.
- `Final` (`'final'`) — The outcome of the pattern search (successful or unsuccessful), the reason for stopping, and the final point.

Both `Iterative` and `Diagnose` display the following information:

- `Iter` — Iteration number
- `FunEval` — Cumulative number of function evaluations
- `MeshSize` — Current mesh size
- `FunVal` — Objective function value of the current point
- `Method` — Outcome of the current poll

The default value of **Level of display** is

- `Off` in the Pattern Search Tool
- `'final'` in an options structure created using `psoptimset`

Vectorize Option

The vectorize option specifies whether the computation of the objective function is vectorized. When you set **Objective function is vectorized** (`'Vectorize'`) to `Off` (`'off'`), the algorithm computes the objective function values of the mesh points in a loop, calling the objective function with exactly one point each time through the loop. On the other hand, when you set **Objective function is vectorized** to `On` (`'on'`), the pattern search algorithm computes the objective function values of all mesh points with a single call to the objective function, which is faster than computing them in a loop. However,

to use this option, your objective function must be able to accept input matrices with an arbitrary number of rows.

Functions — Alphabetical List

This section contains function reference pages listed alphabetically.

Purpose

Find the minimum of a function using the genetic algorithm

Syntax

```
x = ga(fitnessfun, nvars)
x = ga(fitnessfun, nvars, options)
x = ga(problem)
[x, fval] = ga(...)
[x, fval, reason] = ga(...)
[x, fval, reason, output] = ga(...)
[x, fval, reason, output, population] = ga(...)
[x, fval, reason, output, population, scores] = ga(...)
```

Description

`ga` implements the genetic algorithm at the command line to minimize an objective function.

`x = ga(fitnessfun, nvars)` applies the genetic algorithm to an optimization problem, where `fitnessfun` is the objective function to minimize and `nvars` is the length of the solution vector `x`, the best individual found.

`x = ga(fitnessfun, nvars, options)` applies the genetic algorithm to an optimization problem, using the parameters in the `options` structure.

`x = ga(problem)` finds the minimum for `problem`, a structure that has three fields:

- `fitnessfcn` — Fitness function
- `nvars` — Number of independent variables for the fitness function
- `options` — Options structure created with `gaoptimset`

`[x, fval] = ga(...)` returns `fval`, the value of the fitness function at `x`.

`[x, fval, reason] = ga(...)` returns `reason`, a string containing the reason the algorithm stops.

`[x, fval, reason, output] = ga(...)` returns `output`, a structure that contains output from each generation and other information about the performance of the algorithm. The `output` structure contains the following fields:

- `randstate` — The state of `rand`, the MATLAB random number generator, just before the algorithm started.

- randnstate — The state of randn the MATLAB normal random number generator, just before the algorithm started. You can use the values of randstate and randnstate to reproduce the output of ga. See "Reproducing Your Results" on page 4-25.
- generations — The number of generations computed
- funccount — The number of evaluations of the fitness function
- message — The reason the algorithm terminated. This message is the same as the output argument reason.

[x, fval, reason, output, population] = ga(...) returns matrix population, whose rows are the final population.

[x, fval, reason, output, population, scores] = ga(...) returns scores, the scores of the final population.

Note For problems that use the population type Double Vector (the default), ga does not accepts functions whose inputs are of type complex. To solve problems involving complex data, write your functions so that they accept real vectors, by separating the real and imaginary parts.

Example

```
[x fval, reason] = ga(@rastriginsFcn, 10)
x =

  Columns 1 through 7

   0.9977   0.9598   0.0085   0.0097  -0.0274  -0.0173   0.9650

  Columns 8 through 10

   -0.0021   -0.0210    0.0065

fval =

    3.7456
```

```
reason =

generations
```

Reference

[1] Goldberg, David E., *Genetic Algorithms in Search, Optimzation & Machine Learning*, Addison-Wesley, 1989.

See Also

`gaoptimset`, `gatool`

gaoptimget

Purpose

Get values of a genetic algorithm `options` structure

Syntax

```
val = gaoptimget(options, 'name')
```

Description

`val = gaoptimget(options, 'name')` returns the value of the parameter `name` from the genetic algorithm options structure `options`.
`gaoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `gaoptimget` ignores case in parameter names.

See Also

`ga`, `gaoptimset`, `gatool`

Purpose

Create a genetic algorithm options structure

Syntax

```
options = gaoptimset
gaoptimset
options = gaoptimset('param1',value1,'param2',value2,...)
options = gaoptimset(oldopts,'param1',value1,...)
options = gaoptimset(oldopts,newopts)
```

Description

`options = gaoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for the genetic algorithm and sets parameters to their default values.

`gaoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = gaoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of `'param1'` to `value1`, `'param2'` to `value2`, and so on. Any unspecified parameters are set to their default values. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`options = gaoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = gaoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the options you can set with `gaoptimset`. See "Genetic Algorithm Options" on page 6-3 for a complete description of these options and their values. Values in `{}` denote the default value. You can also view the optimization parameters and defaults by typing `gaoptimset` at the command line.

Option	Description	Values
`CreationFcn`	Handle to the function that creates the initial population	`{@gacreationuniform}`
`CrossoverFraction`	The fraction of the population at the next generation, not including elite children, that is created by the crossover function	Positive scalar | `{0.8}`

`CrossoverFcn`	Handle to the function that the algorithm uses to create crossover children	`@crossoverheuristic`
		`{@crossoverscattered}`
		`@crossoverintermediate`
		`@crossoversinglepoint`
		`@crossovertwopoint`

`EliteCount`	Positive integer specifying how many individuals in the current generation are guaranteed to survive to the next generation	Positive integer | `{2}`
`FitnessLimit`	Scalar. If the fitness function attains the value of `FitnessLimit`, the algorithm halts.	Scalar | `{-Inf}`

`FitnessScalingFcn`	Handle to the function that scales the values of the fitness function	`@fitscalinggoldberg`
		`{@fitscalingrank}`
		`@fitscalingprop`
		`@fitscalingtop`

Option	Description	Values
`Generations`	Positive integer specifying the maximum number of iterations before the algorithm halts	Positive integer |`{100}`
`PopInitRange`	Matrix or vector specifying the range of the individuals in the initial population	Matrix or vector | `[0;1]`
`PopulationType`	String describing the data type of the population	`'bitstring'` | `'custom'` | `{'doubleVector'}`
`HybridFcn`	Handle to a function that continues the optimization after ga terminates	Function handle | `{[]}`
`InitialPopulation`	Initial population	Positive scalar | `{[]}`
`InitialScores`	Initial scores	Column vector | `{[]}`
`MigrationDirection`	Direction of migration	`'both'` | `{'forward'}`
`MigrationFraction`	Scalar between 0 and 1 specifying the fraction of individuals in each subpopulation that migrates to a different subpopulation	Scalar | `{0.2}`

Option	Description	Values
`MigrationInterval`	Positive integer specifying the number of generations that take place between migrations of individuals between subpopulations	Positive integer | `{20}`
`MutationFcn`	Handle to the function that produces mutation children	`@mutationuniform` `{@mutationgaussian}`
`OutputFcns`	Array of handles to functions that `ga` calls at each iteration.	Array | `{[]}`
`OutputInterval`	Positive integer specifying the number of generations between consecutive calls to the output functions	Positive integer | `{1}`

`PlotFcns`	Array of handles to functions that plot data computed by the algorithm	`@gaplotbestf`
		`@gaplotbestgenome`
		`@gaplotdistance`
		`@gaplotexpectation`
		`@gaplotgeneology`
		`@gaplotselection`
		`@gaplotrange`
		`@gaplotscorediversity`
		`@gaplotscores`
		`@gaplotstopping` | `{[]}`

`PlotInterval`	Positive integer specifying the number of generations between consecutive calls to the plot functions	Positive integer | `{1}`

Option	Description	Values
`PopulationSize`	Size of the population	Positive integer | `{20}`

`SelectionFcn`	Handle to the function that selects parents of crossover and mutation children	`@selectiongoldberg`
		`@selectionrandom`
		`{@selectionstochunif}`
		`@selectionroulette`
		`@selectiontournament`

`StallGenLimit`	Positive integer. The algorithm stops if there is no improvement in the objective function for `StallGenLimit` consecutive generations.	Positive integer | `{50}`
`StallTimeLimit`	Positive scalar. The algorithm stops if there is no improvement in the objective function for `StallTimeLimit` seconds.	Positive scalar | `{20}`
`TimeLimit`	Positive scalar. The algorithm stops after running for `TimeLimit` seconds.	Positive scalar | `{30}`
`Vectorized`	String specifying whether the computation of the fitness function is vectorized	`'on' | {'off'}`

See Also

`gaoptimget`, `gatool`

Purpose

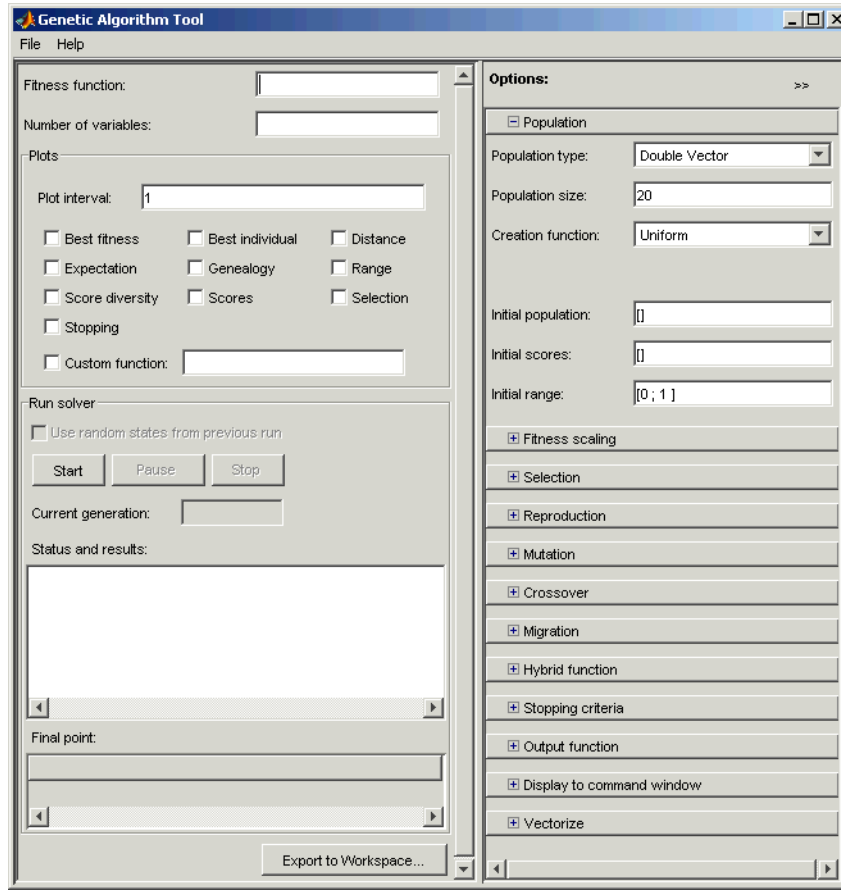
Open the Genetic Algorithm Tool

Syntax

```
gatool
```

Description

`gatool` opens the Genetic Algorithm Tool, a graphical user interface (GUI) to the genetic algorithm, as shown in the figure below.



You can use the Genetic Algorithm Tool to run the genetic algorithm on optimization problems and display the results. See “Using the Genetic Algorithm Tool” on page 2-4 for a complete description of the tool.

See Also

`ga`, `gaoptimset`

patternsearch

Purpose

Find the minimum of a function using a pattern search

Syntax

```
x = patternsearch(@fun, x0)
x = patternsearch(@fun, x0, A, b)
x = patternsearch(@fun, x0, A, b, Aeq,  beq)
x = patternsearch(@fun, x0, A, b, Aeq,  beq, lb, ub)
x = patternsearch(@fun, x0, A, b, Aeq,  beq, lb, ub, options)
x = patternsearch(problem)
[x, fval] = patternsearch(@fun, x0, ...)
[x, fval, exitflag] = patternsearch(@fun, x0, ...)
[x, fval, exitflag, output] = patternsearch(@fun, x0, ...)
```

Description

`patternsearch` finds the minimum of a function using a pattern search.

`x = patternsearch(@fun, x0)` solves unconstrained problems of the form

$$\underset{x}{\text{minimize}}\ f(x)$$

where `fun` is a MATLAB function that computes the values of the objective function $f(x)$, and `x0` is an initial point for the pattern search algorithm. The function `patternsearch` accepts the objective function as a function handle of the form `@fun`. `patternsearch` returns a local minimum `x` to the objective function. The function `fun` accepts a vector input and returns a scalar function value.

`x = patternsearch(@fun, x0, A, b)` finds a local minimum `x` to the function `fun`, subject to the linear inequality constraints represented in matrix form by

$$A x \le b$$

If the problem has m linear inequality constraints and n variables, then

- `A` is a matrix of size m-by-n.
- `b` is a vector of length m.

`x = patternsearch(@fun, x0, A, b, Aeq, beq)` finds a local minimum `x` to the function `fun`, subject to the constraints

$$A x \le b$$
$$Aeq\ x = beq$$

where $Aeq\ x = beq$ represents the linear equality constraints in matrix form. If the problem has r linear equality constraints and n variables, then

- `Aeq` is a matrix of size r-by-n.
- `beq` is a vector of length r.

If there are no inequality constraints, pass empty matrices, `[]`, for `A` and `b`.

`x = patternsearch(@fun, x0, A, b, Aeq, beq, lb, ub)` finds a local minimum `x` to the function `fun` subject to the constraints

$$A x \le b$$
$$Aeq\ x = beq$$
$$lb \le x \le ub$$

where $lb \le x \le ub$ represents lower and upper bounds on the variables. If the problem has n variables, `lb` and `ub` are vectors of length n. If `lb` or `ub` is empty (or not provided), it is automatically expanded to `-Inf` or `Inf`, respectively. If there are no inequality or equality constraints, pass empty matrices for `A`, `b`, `Aeq` and `beq`.

`x = patternsearch(@fun, x0, A, b, Aeq, beq, lb, ub, options)` finds a local minimum `x` to the function `fun`, replacing the default optimization parameters by values in the structure `options`. You can create `options` with the function `psoptimset`. Pass empty matrices for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, and `options` to use the default values.

`x = patternsearch(problem)` finds the minimum for `problem`, a structure that has the following fields:

- `objective` — Objective function
- `X0` — Starting point
- `Aineq` — Matrix for the inequality constraints
- `Bineq` — Vector for the inequality constraints

- `Aeq` — Matrix for the equality constraints
- `Beq` — Vector for the equality constraints
- `LB` — Lower bound for x
- `UB` — Upper bound for x
- `options` — Options structure created with `psoptimset`
- `randstate` — Optional field to reset the state of `rand`
- `randnstate` — Optional field to reset the state of `randn`

You can create the structure `problem` by exporting a problem from the Pattern Search Tool, as described in "Importing and Exporting Options and Problems" on page 5-11.

Note `problem` must have all the fields as specified above.

`[x, fval] = patternsearch(@fun, x0, ...)` returns the value of the objective function `fun` at the solution `x`.

`[x, fval, exitflag] = patternsearch(@fun, x0, ...)` returns `exitflag`, which describes the exit condition of `patternsearch`. If

- `exitflag` > 0, `patternsearch` converged to a solution `x`.
- `exitflag = 0`, `patternsearch` reached the maximum number of function evaluations or iterations.
- `exitflag < 0`, `patternsearch` did not converge to a solution.

`[x, fval, exitflag, output] = patternsearch(@fun, x0, ...)` returns a structure `output` containing information about the search. The output structure contains the following fields:

- `function` — Objective function
- `problemtype` — Type of problem: unconstrained, bound constrained or linear constrained
- `pollmethod` — Polling method
- `searchmethod` — Search method used, if any
- `iteration` — Total number of iterations

- `funccount` — Total number of function evaluations
- `meshsize` — Mesh size at `x`
- `message` — Reason why the algorithm terminated

Note `patternsearch` does not accepts functions whose inputs are of type `complex`. To solve problems involving complex data, write your functions so that they accept real vectors, by separating the real and imaginary parts.

Example

Given the following constraints

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 \le x_1 \\ 0 \le x_2 \end{bmatrix}$$

the following code finds the minimum of the function, `lincontest6`, that is provided with the toolbox:

```
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
lb = zeros(2,1);
[x, fval, exitflag] = patternsearch(@lincontest6,...
[0 0],A,b,[],[],lb)
Optimization terminated:
Next Mesh size (9.5367e-007)less than 'TolMesh.'

x =

    0.6667    1.3333
```

```
fval =

   -8.2222


exitflag =

     1
```

References

[1] Torczon, Virginia, "On the convergence of Pattern Search Algorithms," SIAM Journal on Optimization, Vol. 7, Number 1, pp. 1-25, 1997.

[2] Lewis, Robert Michael and Virginia Torczon, "Pattern Search Algorithms for Bound Constrained Minimization," SIAM Journal on Optimization, Vol. 9, Number 4, pp. 1082-1099, 1999.

[3] Lewis, Robert Michael and Virginia Torczon, "Pattern Search Methods for Linearly Constrained Minimization," SIAM Journal on Optimization, Vol. 10, Number 3, pp. 917-941, 2000.

[4] Audet, Charles and J.E. Dennis Jr., "Analysis of Generalized Pattern Searches," SIAM Journal on Optimization, Vol. 13, Number 3, pp. 889-903, 2003.

See Also

`psearchtool`, `psoptimget`, `psoptimset`

Purpose

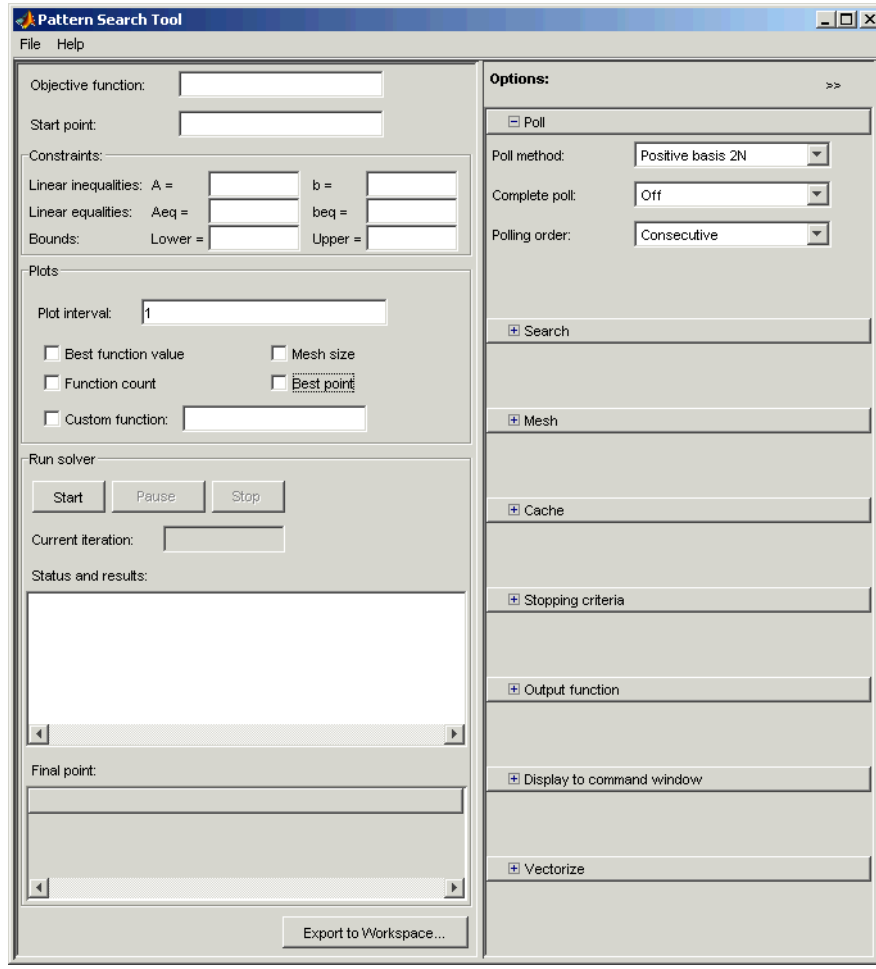
Open the Pattern Search Tool

Syntax

```
psearchtool
```

Description

`psearchtool` opens the Pattern Search Tool, a graphical user interface (GUI) for performing pattern searches, as shown in the figure below.



You can use the Pattern Search Tool to run a pattern search on optimization problems and display the results. See “Using the Pattern Search Tool” on page 3-3 for a complete description of the tool.

See Also

`patternsearch`, `psoptimget`, `psoptimset`

Purpose

Get values of a pattern search options structure

Syntax

```
val = psoptimget(options, 'name')
```

Description

`val = psoptimget(options, 'name')` returns the value of the parameter `name` from the pattern search options structure `options`.
`psoptimget(options, 'name')` returns an empty matrix `[]` if the value of `name` is not specified in `options`. It is only necessary to type enough leading characters of `name` to uniquely identify it. `psoptimget` ignores case in parameter names.

See Also

`psoptimset`, `patternsearch`

psoptimset

Purpose

Create a pattern search options structure

Syntax

```
options = psoptimset
psoptimset
options = psoptimset('param1',value1,'param2',value2,...)
options = psoptimset(oldopts,'param1',value1,...)
options = psoptimset(oldopts,newopts)
```

Description

`options = psoptimset` (with no input arguments) creates a structure called `options` that contains the options, or *parameters*, for the pattern search and sets parameters to their default values.

`psoptimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = psoptimset('param1',value1,'param2',value2,...)` creates a structure `options` and sets the value of `'param1'` to `value1`, `'param2'` to `value2`, and so on. Any unspecified parameters are set to their default values. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`options = psoptimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = psoptimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the options you can set with `psoptimset`. See "Pattern Search Options" on page 6-21 for a complete description of the options and their values. Values in `{}` denote the default value. You can also view the optimization parameters and defaults by typing `psoptimset` at the command line.

Option	Description	Values
`Cache`	With `Cache` set to `'on'`, `patternsearch` keeps a history of the mesh points it polls and does not poll points close to them again at subsequent iterations. Use this option if `patternsearch` runs slowly because it is taking a long time to compute the objective function.	`'on' | {'off'}`
`CacheSize`	Size of the history	Positive scalar | `{1e4}`
`CacheTol`	Positive scalar specifying how close the current mesh point must be to a point in the history in order for `patternsearch` to avoid polling it	Positive scalar | `{1e-10}`
`CompletePoll`	Complete poll around current iterate	`'on' | {'off'}`
`CompleteSearch`	Complete poll around current iterate	`'on' | {'off'}`
`Display`	Level of display	`'off' |'iter' | 'notify' | 'diagnose' | {'final'}`
`InitialMeshSize`	Initial mesh size for pattern algorithm	Positive scalar | `{1.0}`

`MaxFunEvals`	Maximum number of objective function evaluations	Positive integer | `{2000*numberOfVariables}`
`MaxIter`	Maximum number of iterations	Positive integer | `{100*numberOfVariables}`
`MaxMeshSize`	Maximum mesh size	Positive scalar | `{Inf}`
`MeshAccelerator`	Accelerate convergence near a minimum	`'on' | {'off'}`
`MeshContraction`	Mesh contraction factor. Used when iteration is unsuccessful.	Positive scalar | `{0.5}`
`MeshExpansion`	Mesh expansion factor. Expands mesh when iteration is successful.	Positive scalar |`{2.0}`
`OutputFcn`	Specifies a user-defined function that an optimization function calls at each iteration	`@psoutputhistory | {none}`
`PlotFcn`	Specifies plots of output from the pattern search	`@psplotbestf | @psplotmeshsize | @psplotfuncount | {[]}`
`PlotInterval`	Specifies the number of iterations between consecutive calls to the plot functions	Positive integer
`PollingOrder`	Order of poll directions in pattern search	`'Random' | 'Success' | {'Consecutive'}`
`PollMethod`	Polling strategy used in pattern search	`{'PositiveBasis2N'} | 'PositiveBasisNp1'`
`ScaleMesh`	Automatic scaling of variables	`{'on'} | 'off'`

SearchMethod	Type of search used in pattern search	'PositiveBasisNp1' | 'PositiveBasis2N' | @searchga | @searchlhs | @searchneldermead| {[]}
TolBind	Binding tolerance	Positive scalar | {1e-3}
TolCon	Tolerance on constraints	Positive scalar | {1e-6}
TolFun	Tolerance on function	Positive scalar | {1e-6}
TolMesh	Tolerance on mesh size	Positive scalar | {1e-6}
TolX	Tolerance on variable	Positive scalar | {1e-6}
Vectorized	Specifies whether functions are vectorized	'on' | {'off'}

For a detailed description of these options, see “Pattern Search Options” on page 6-21.

See Also

patternsearch, psoptimget

Index

A

accelerator, mesh 5-33
algorithm
  genetic 2-18
  pattern search 3-13

C

cache 5-34
children 2-17
  crossover 2-20
  elite 2-20
  mutation 2-20
crossover 4-40
  children 2-20
  fraction 4-43

D

direct search 3-2
diversity 2-16

E

elite children 2-20
expansion, mesh 5-28
exporting problems
  from Genetic Algorithm Tool 4-13
  from Pattern Search Tool 5-11

F

fitness function 2-15
  vectorizing 4-56
  writing M-files for 1-3
fitness scaling 4-35

G

`ga` 6-37
`gaoptimget` 6-40
`gaoptimset` 6-41
`gatool` 6-46
generation 2-15
genetic algorithm
  description of 2-18
  options 6-3
  overview 2-2
  setting options at command line 4-22
  stopping criteria for 2-23
  using from command line 4-21
Genetic Algorithm Tool 4-2
  defining a problem in 4-3
  displaying plots 4-7
  exporting options and problems from 4-13
  importing problems to 4-16
  opening 4-2
  pausing and stopping 4-6
  running 4-4
  setting options in 4-12
global and local minima 4-48

H

hybrid function 4-52

I

importing problems
  to Genetic Algorithm Tool 4-16
  to Pattern Search Tool 5-13
individual 2-15
initial population 2-19

M

maximizing 1-4
mesh 3-11
  accelerator 5-33
  expansion and contraction 5-28
M-files, writing 1-3
minima, global and local 4-48
minimizing 1-4
mutation 4-40
  options 4-41

O

objective function
  writing M-file for 1-3
options
  genetic algorithm 6-3

P

parent 2-17
pattern 3-10
pattern search
  description 3-13
  options 6-21
  overveiw 3-2
  setting options at command line 5-16
  using from command line 5-14
Pattern Search Tool 5-2
  defining problem in 5-3
  displaying plots 5-8
  exporting options and problems from 5-11
  importing problems to 5-13
  opening 5-2
  pausing and stopping 5-8
  running 5-5
  setting options in 5-10
`patternsearch` 6-48
plots
  genetic algorithm 4-7
  pattern search 5-8
poll 3-12
  complete 5-21
  method 5-19
population 2-15
  initial 2-19
  initial range 4-31
  options 4-30
  size 4-34
`psearchtool` `6-53`
`psoptimget` 6-55
`psoptimset` 6-56

R

Rastrigin's function 2-6
reproduction 4-40

S

scaling, fitness 4-35
search method 5-25
selection 4-39
setting options
  genetic algorithm 4-30
  pattern search 5-19

stopping criteria
pattern search 3-18

V

vectorizing 4-56