Genetic Algorithm and Direct Search Toolbox™ 2
User's Guide

MATLAB®

The MathWorks™

Accelerating the pace of engineering and science
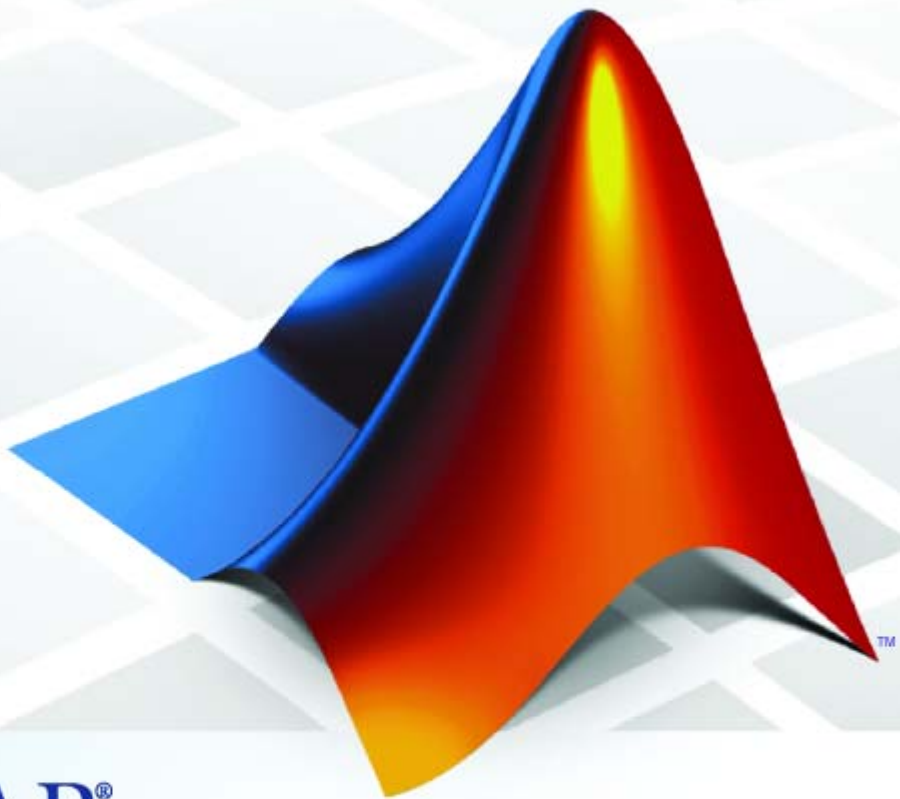
How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	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.
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

Trademarks

Patents

Revision History

Acknowledgment

Acknowledgment

Contents

Getting Started with the Genetic Algorithm

3

Getting Started with Simulated Annealing and Threshold Acceptance

4

Using the Genetic Algorithm

6

Using the Simulated Annealing and Threshold Acceptance Algorithms

7

Multiobjective Optimization

8

<div align="right">

Options Reference

</div>

9

Function Reference

10

Functions — Alphabetical List

11

Examples

A

Index

Introducing Genetic Algorithm and Direct Search Toolbox™ Functions

Product Overview (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.

Product Overview

Genetic Algorithm and Direct Search Toolbox™ functions extend the capabilities of Optimization Toolbox™ software and the MATLAB® numeric computing environment. They include routines for solving optimization problems using

- Direct search
- Genetic algorithm
- Simulated annealing

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

All the toolbox functions are 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 Genetic Algorithm and Direct Search Toolbox functions by writing your own M-files, or by using them in combination with other toolboxes, or with the MATLAB or Simulink® environments.

Writing M-Files for Functions You Want to Optimize

Computing Objective Functions

To use Genetic Algorithm and Direct Search Toolbox™ functions, you must first write an M-file (or else an anonymous function) that computes the function you want to optimize. The M-file should accept a 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.

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 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** from 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
```

Maximizing vs. Minimizing

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

$$\min_x f(x).$$

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

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

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

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

and minimize $g(x)$.

2

Getting Started with Direct 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. Unlike 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 is not even continuous.

Genetic Algorithm and Direct Search Toolbox™ functions include two direct search algorithms called the generalized pattern search (GPS) algorithm and the mesh adaptive search (MADS) algorithm. Both are *pattern search* algorithms that compute a sequence of points that approach an 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 mesh is formed by adding the current point to a scalar multiple of a set of vectors called a *pattern*. If the pattern search 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.

The MADS algorithm is a modification of the GPS algorithm. The algorithms differ in how the set of points forming the mesh is computed. The GPS algorithm uses fixed direction vectors, whereas the MADS algorithm uses a random selection of vectors to define the mesh.

Performing a Pattern Search

In this section...

Calling patternsearch at the Command Line

To perform a pattern search on an unconstrained problem at the command line, 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:

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

"Performing a Pattern Search from the Command Line" on page 5-11 explains in detail how to use the patternsearch function.

Using the Optimization Tool for Pattern Search

To open the Optimization Tool, enter

```
optimtool('patternsearch')
```

at the command line, or enter optimtool and then choose patternsearch from the **Solver** menu.

Set options

Expand or contract help

Choose solver

Enter problem
and constraints

Run solver

View results

See final point
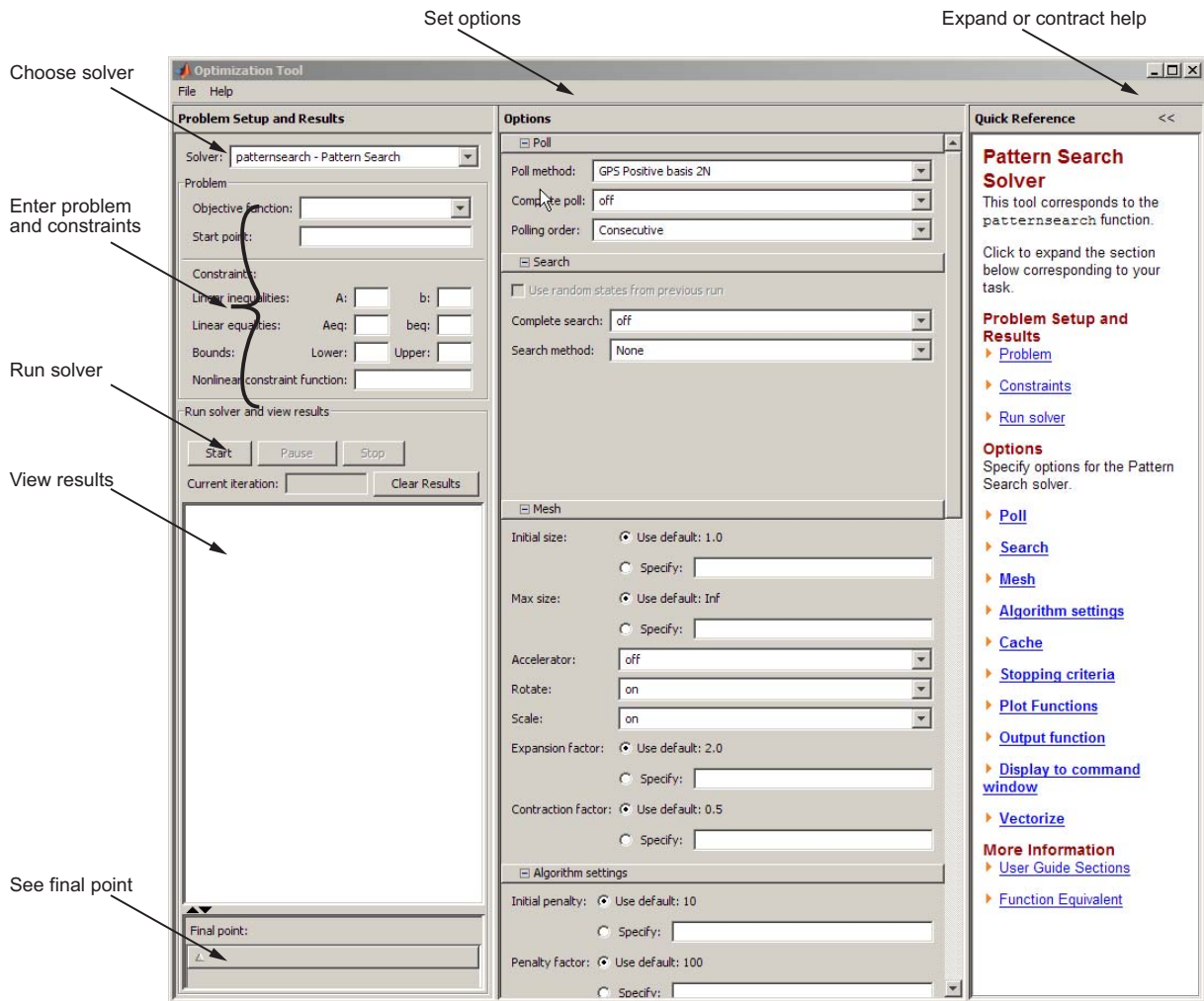
You can also launch the tool from the MATLAB® **Start** menu as pictured:
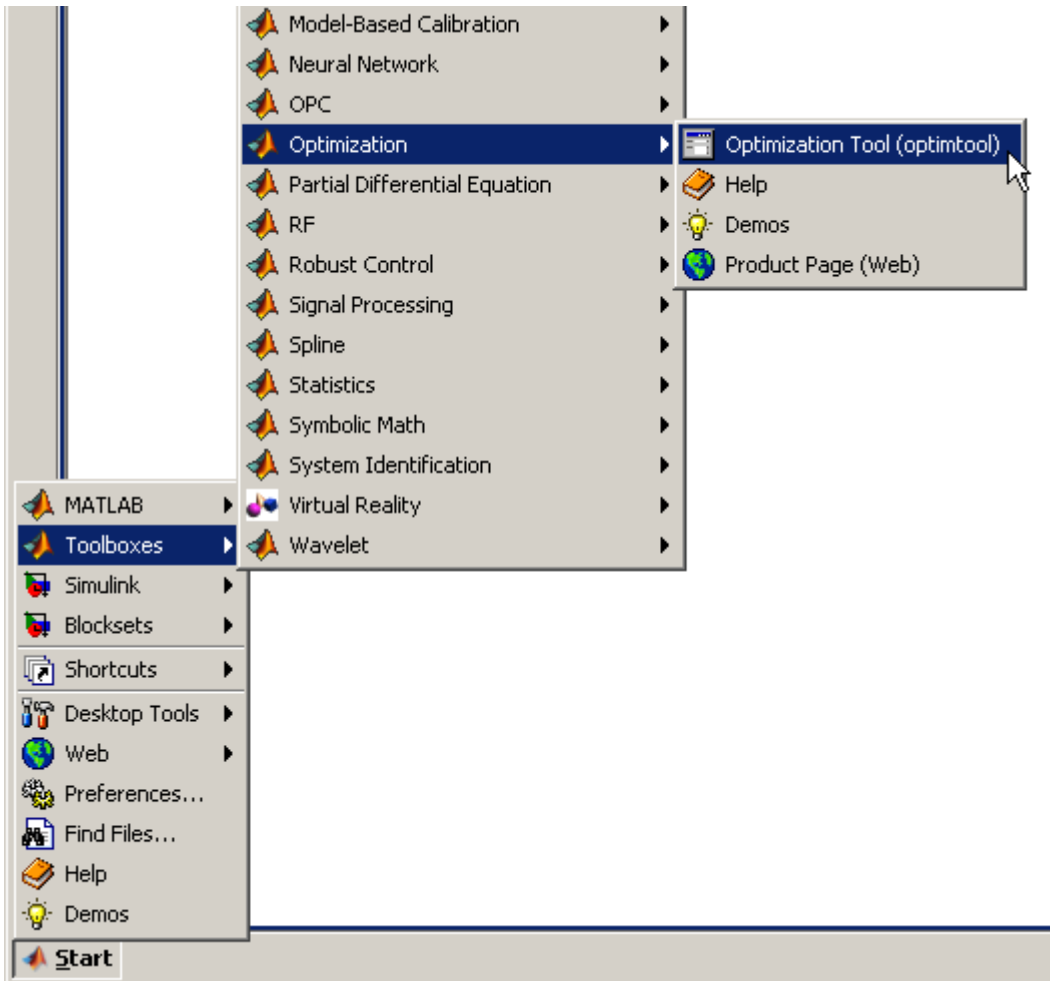
To use the Optimization Tool, first enter the following information:

- **Objective function** — The objective function you want to minimize. 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.

In the **Constraints** pane, enter linear constraints, bounds, or a nonlinear constraint function as a function handle for the problem. If the problem is unconstrained, leave these fields blank.

Then, click **Start**. The tool displays the results of the optimization in the **Run solver and view results** pane.

In the **Options** pane, set the options for the pattern search. To view the options in a category, click the + sign next to it.

"Finding the Minimum of the Function" on page 2-8 gives an example of using the Optimization Tool.

The "Optimization Tool" chapter in the *Optimization Toolbox™ User's Guide* provides a detailed description of the Optimization Tool.

Example: Finding the Minimum of a Function Using the GPS Algorithm

Objective Function

This example uses the objective function, ps_example, which is included with Genetic Algorithm and Direct Search Toolbox™ software. View the code for the function by entering

```
type ps_example
```

The following figure shows a plot of the function.
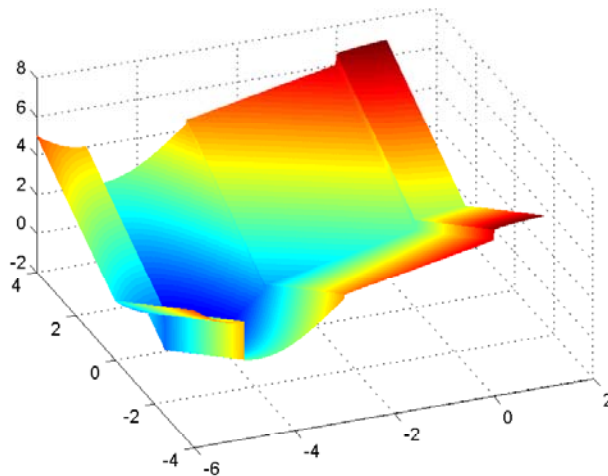
Finding the Minimum of the Function

To find the minimum of ps_example, perform the following steps:

1 Enter

```
optimtool
```

and then choose the patternsearch solver.

2 In the **Objective function** field of the Optimization Tool, enter @ps_example.

3 In the **Start point** field, type [2.1 1.7].
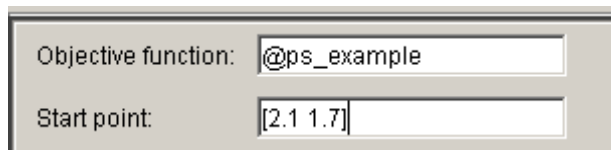
Leave the fields in the **Constraints** pane blank because the problem is unconstrained.

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

The **Run solver and view results** pane displays the results of the pattern search.
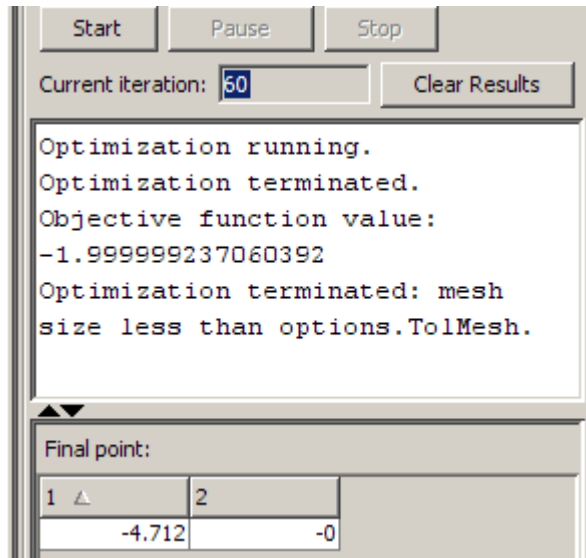
The reason the optimization terminated is that the mesh size became smaller than the acceptable tolerance value for the mesh size, defined by the **Mesh tolerance** parameter in the **Stopping criteria** pane. The minimum function value is approximately –2. The **Final point** pane displays the point at which the minimum occurs.

Plotting the Objective Function Values and Mesh Sizes

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

- **Best function value**
- **Mesh size**
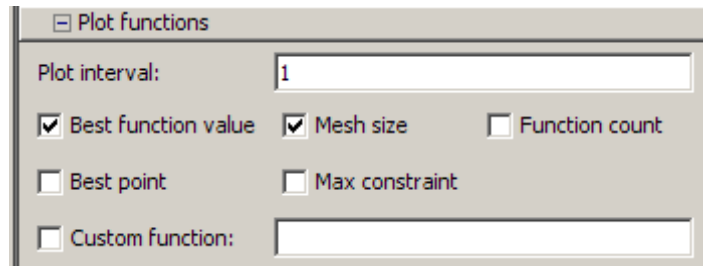
Then click **Start** to run the pattern search. This displays the following plots.
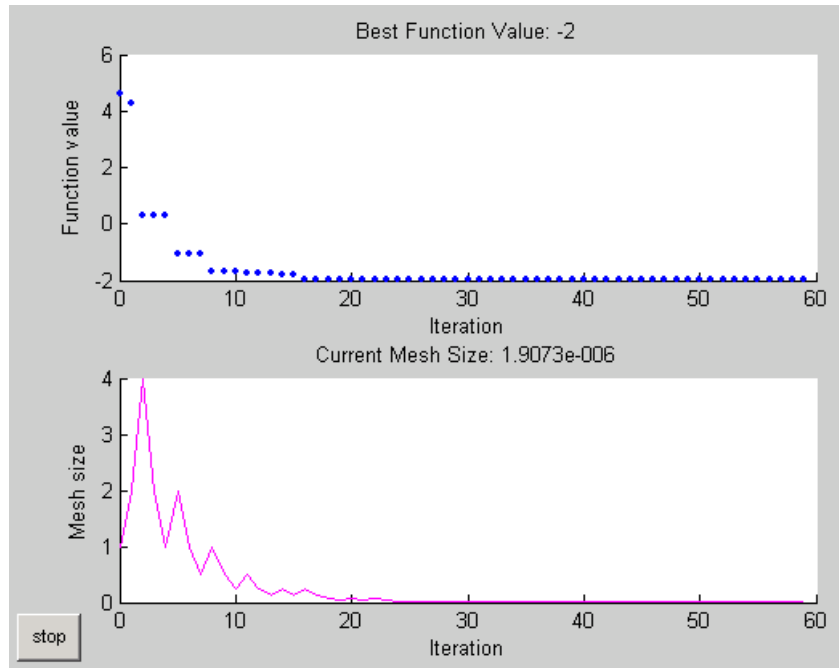
The upper plot shows the objective function value of the best point at each iteration. Typically, the objective function values improve rapidly at the early iterations and then level off as they approach the optimal value.

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 2-15.

Pattern Search Terminology

Patterns

A *pattern* is a set of vectors $\{v_i\}$ that the pattern search algorithm uses to determine which points to search at each iteration. The set $\{v_i\}$ is defined by the number of independent variables in the objective function, N, and the positive basis set. Two commonly used positive basis sets in pattern search algorithms are the maximal basis, with $2N$ vectors, and the minimal basis, with $N+1$ vectors.

With GPS, the collection of vectors that form the pattern are fixed-direction vectors. For example, if there are three independent variables in the optimization problem, the default for a $2N$ positive basis consists of the following pattern vectors:

$$v_1 = [1 \quad 0 \quad 0] \quad v_2 = [0 \quad 1 \quad 0] \quad v_3 = [0 \quad 0 \quad 1]$$
$$v_4 = [-1 \quad 0 \quad 0] \quad v_5 = [0 \quad -1 \quad 0] \quad v_6 = [0 \quad 0 \quad -1]$$

An $N+1$ positive basis consists of the following default pattern vectors.

$$v_1 = [1 \quad 0 \quad 0] \quad v_2 = [0 \quad 1 \quad 0] \quad v_3 = [0 \quad 0 \quad 1]$$
$$v_4 = [-1 \quad -1 \quad -1]$$

With MADS, the collection of vectors that form the pattern are randomly selected by the algorithm. Depending on the poll method choice, the number of vectors selected will be $2N$ or $N+1$. As in GPS, $2N$ vectors consist of N vectors and their N negatives, while $N+1$ vectors consist of N vectors and one that is the negative of the sum of the others.

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 GPS and MADS algorithms form the mesh by

1 Generating a set of vectors $\{d_i\}$ by multiplying each pattern vector v_i by a scalar Δ^m. Δ^m is called the *mesh size*.

2 Adding the $\{d_i\}$ to the *current point*—the point with the best objective function value found at the previous step.

For example, using the GPS algorithm. suppose that:

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

- The pattern consists of the vectors

$$v_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}$$
$$v_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$v_3 = \begin{bmatrix} -1 & 0 \end{bmatrix}$$
$$v_4 = \begin{bmatrix} 0 & -1 \end{bmatrix}$$

- The current mesh size Δ^m 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 the **Complete poll** option 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.

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.

When the **Complete poll** option has the setting 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.

Expanding and Contracting

After polling, the algorithm changes the value of the mesh size Δ^m. The default is to multiply Δ^m by 2 after a successful poll, and by 0.5 after an unsuccessful poll.

How Pattern Search Works

Context

The pattern search algorithms find a sequence of points, x0, x1, x2, ... , that approaches an optimal point. The value of the objective function either decreases or remains the same 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 Using the GPS Algorithm" on page 2-7.

To simplify the explanation, this section describes how the generalized pattern search (GPS) works using a maximal positive basis of 2*N*, with **Scale** set to Off in **Mesh** options.

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 GPS 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.

Objective Function Values at Initial Point and Mesh Points

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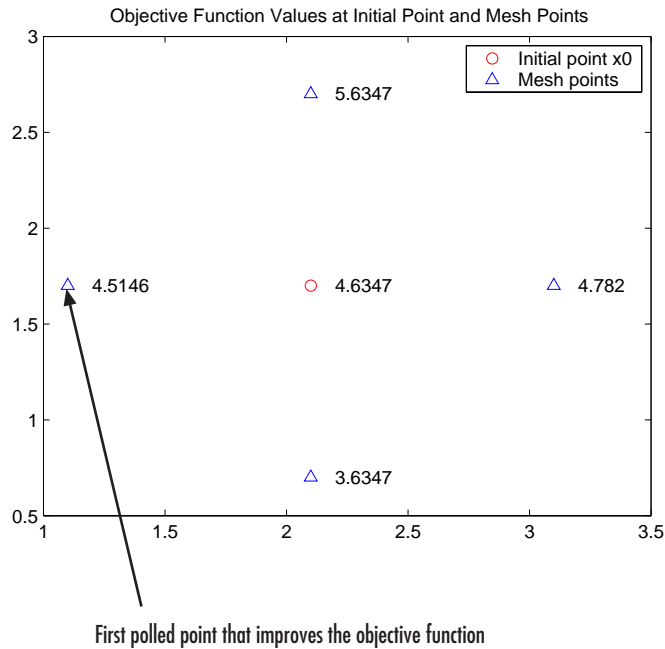
The algorithm polls the mesh points by computing their objective function values until it finds one whose value is smaller than 4.6347, the value at x0. In this case, the first such point it finds is [1.1 1.7], at which the value of the objective function is 4.5146, so the poll at iteration 1 is *successful*.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

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

Note By default, the GPS 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.
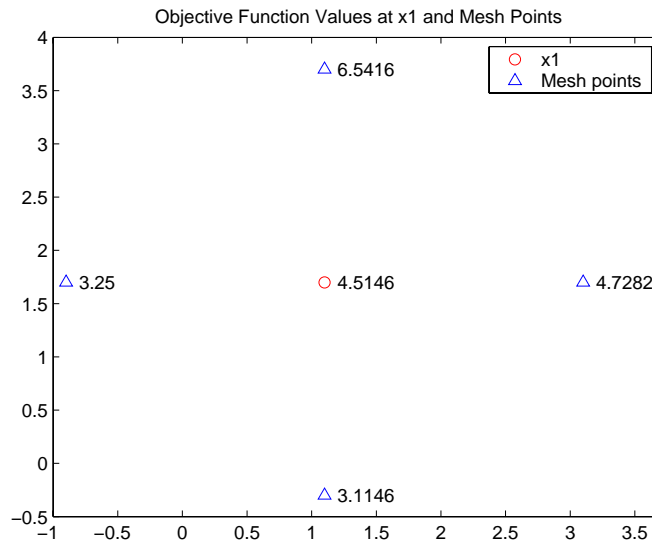
The algorithm polls the mesh points until it finds one whose value is smaller than 4.5146, the value at x1. The first such point it finds is [-0.9 1.7], at which the value of the objective function is 3.25, so the poll at iteration 2 is again successful. The algorithm sets the second point in the sequence 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 = [-4.9 1.7]
```

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

```
8*[1 0] + x3 = [3.1 1.7]
8*[0 1] + x3 = [-4.9 9.7]
8*[-1 0] + x3 = [-12.9 1.7]
```

```
8*[0 -1] + x3 = [-4.9 -1.3]
```

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

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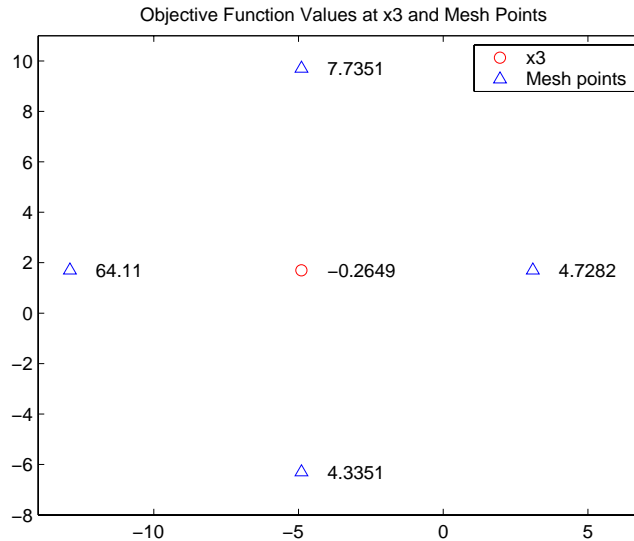
At this iteration, none of the mesh points has a smaller objective function value than the value at x3, so the poll is *unsuccessful*. In this case, the algorithm does not change the current point at the next iteration. That is,

```
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 the **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            f(x)      MeshSize      Method
   0          1         4.63474             1
   1          4         4.51464             2      Successful Poll
   2          7            3.25             4      Successful Poll
   3         10       -0.264905             8      Successful Poll
```

The entry Successful Poll below Method indicates that the current iteration was successful. For example, the poll at iteration 2 is 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 3, 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.

By default, the pattern search doubles the mesh size after each successful poll and halves it after each unsuccessful poll.

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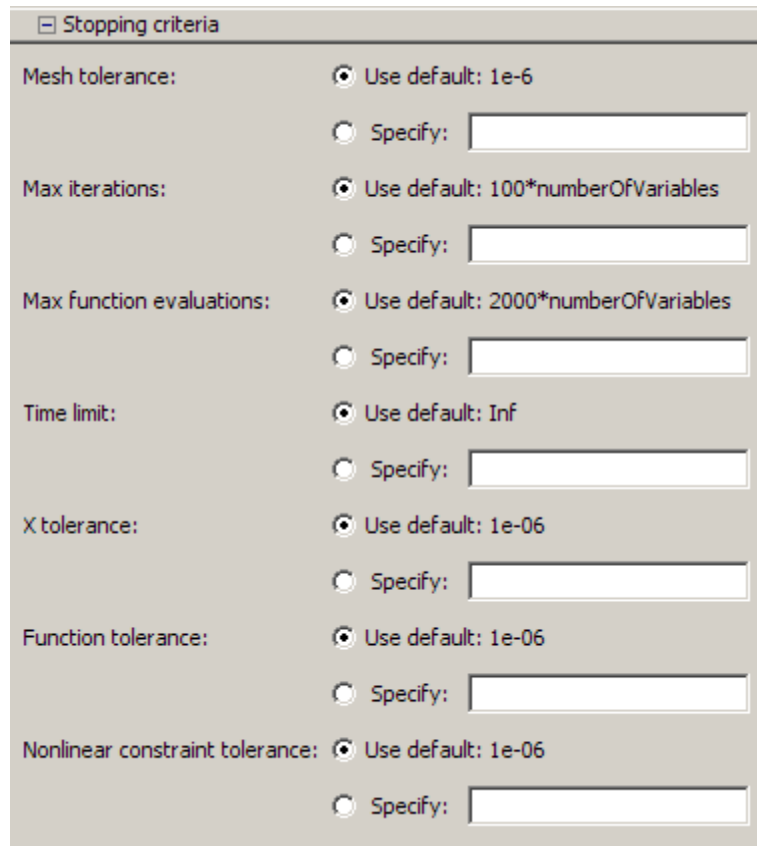
The pattern search performs 88 iterations before stopping. The following plot shows the points in the sequence computed in the first 13 iterations of the pattern search.

Points at First 13 Iterations of Pattern Search
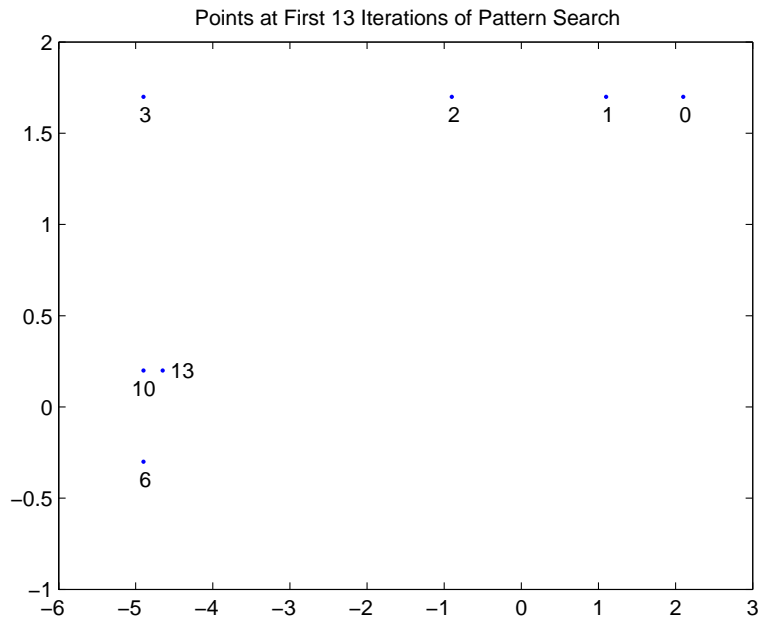
The numbers below the points indicate the first iteration at which the algorithm finds the point. The plot only shows iteration numbers corresponding to successful polls, because the best point doesn't change after an unsuccessful poll. For example, the best point at iterations 4 and 5 is the same as at iteration 3.

Stopping Conditions for the Pattern Search

The criteria for stopping the pattern search algorithm are listed in the **Stopping criteria** section of the Optimization Tool:

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 time, in seconds, the algorithm runs until it reaches the value of **Time limit**.

- The distance between the point found in two consecutive iterations and the mesh size are both less than **X tolerance**.

- The change in the objective function in two consecutive iterations and the mesh size are both less than **Function tolerance**.

Nonlinear constraint tolerance is not used as stopping criterion. It determines the feasibility with respect to nonlinear constraints.

The MADS algorithm uses the relationship between the mesh size, Δ^m, and an additional parameter, called the poll parameter, Δ^p, to determine the stopping criteria. For positive basis $N+1$, the poll parameter is $N\sqrt{\Delta^m}$, and for positive basis $2N$, the poll parameter is $\sqrt{\Delta^m}$. The relationship for MADS stopping criteria is $\Delta^m \leq$ **Mesh tolerance**.

Description of the Nonlinear Constraint Solver

The pattern search algorithm uses the Augmented Lagrangian Pattern Search (ALPS) algorithm to solve nonlinear constraint problems. The optimization problem solved by the ALPS algorithm is

$$\min_{x} f(x)$$

such that

$$
\begin{aligned}
c_i(x) &\leq 0,\ i = 1 \ldots m \\
ceq_i(x) &= 0,\ i = m+1 \ldots mt \\
A \cdot x &\leq b \\
Aeq \cdot x &= beq \\
lb &\leq x \leq ub,
\end{aligned}
$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The ALPS algorithm attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the objective function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using a pattern search algorithm such that the linear constraints and bounds are satisfied.

A subproblem formulation is defined as

$$\Theta(x,\lambda,s,\rho) = f(x) - \sum_{i=1}^{m} \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i c_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} c_i(x)^2,$$

where

- the components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates

- the elements s_i of the vector s are nonnegative shifts

- ρ is the positive penalty parameter.

The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The pattern search algorithm minimizes a sequence of the subproblem, which is an approximation of the original problem. When the subproblem is minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met.

For a complete description of the algorithm, see the following references:

[1] Lewis, Robert Michael and Virginia Torczon, "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Optimization*, Volume 12, Number 4, 2002, 1075–1089.

[2] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, 1991, 545–572.

[3] Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *Mathematics of Computation*, Volume 66, Number 217, 1997, 261–288.

3

Getting Started with the Genetic Algorithm

What Is the Genetic Algorithm?

The genetic algorithm is a method for solving both constrained and unconstrained 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 to 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 classical, derivative-based, optimization algorithm in two main ways, as summarized in the following table.

Classical 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 best point in the population approaches an optimal solution.
Selects the next point in the sequence by a deterministic computation.	Selects the next population by computation which uses random number generators.

Performing a Genetic Algorithm Optimization

In this section...

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

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

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 6-12 provides a detailed description of using the function ga and creating the options structure.

Using the Optimization Tool

To open the Optimization Tool, enter

```
optimtool('ga')
```

at the command line, or enter `optimtool` and then choose ga from the **Solver** menu.
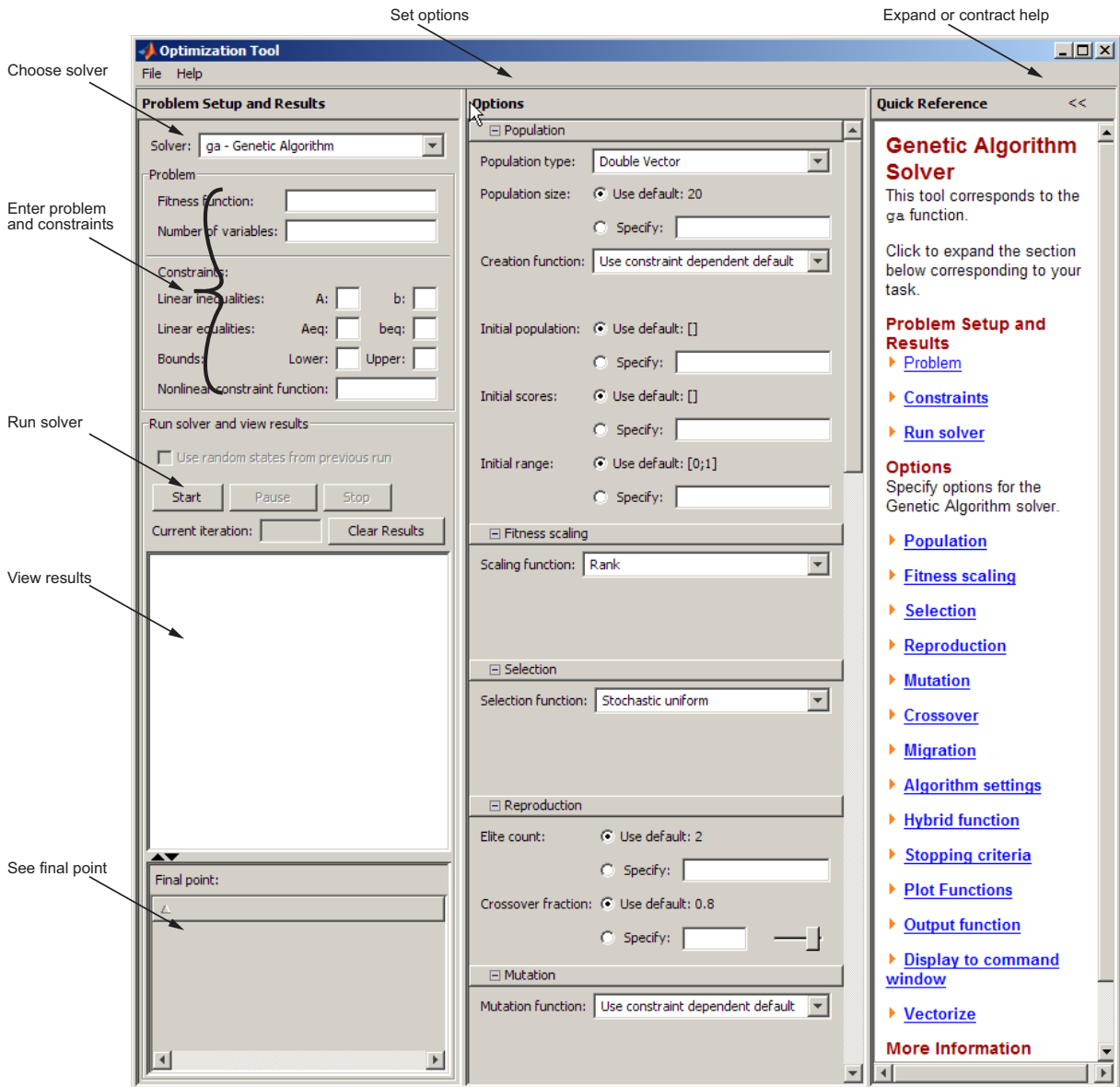
Set options

Expand or contract help

Choose solver

Enter problem and constraints

Run solver

View results

See final point

You can also launch the tool from the MATLAB **Start** menu as pictured:
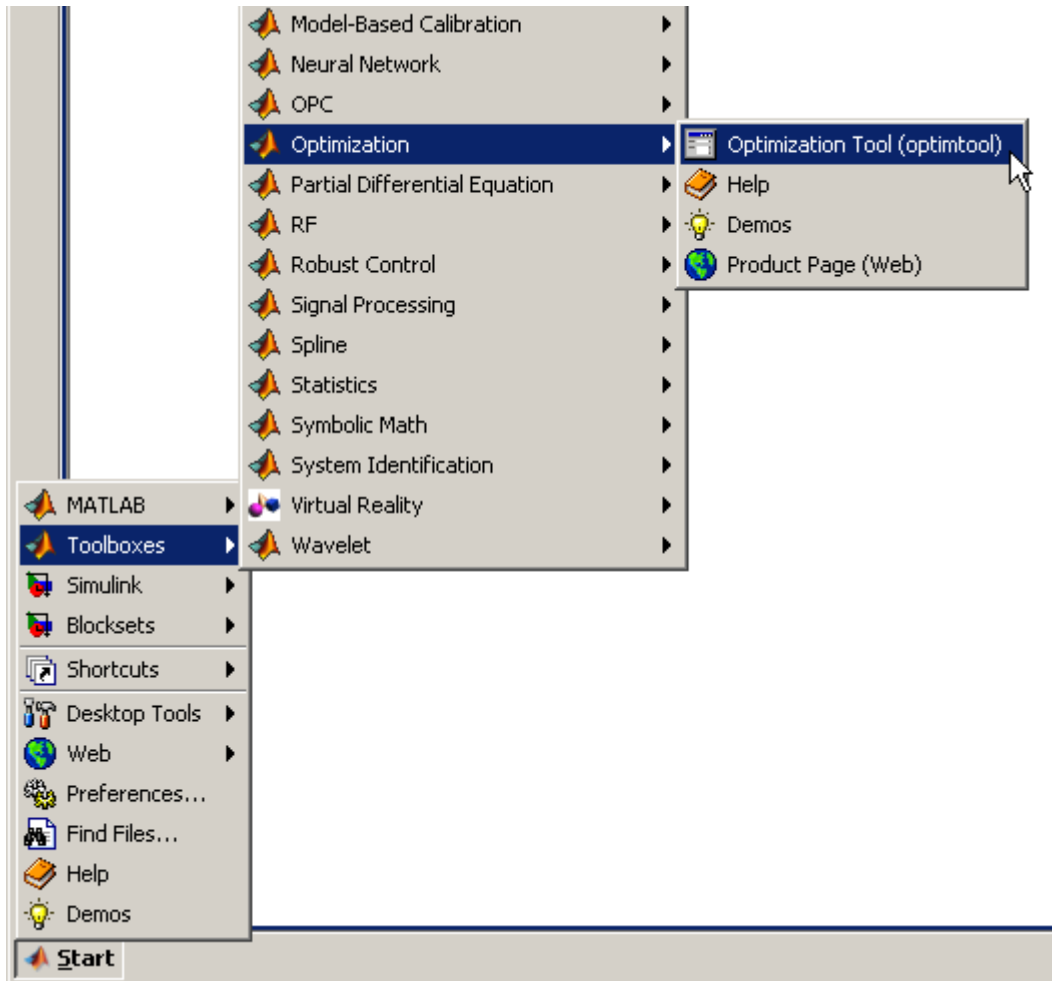
To use the Optimization Tool, you must first enter the following information:

- **Fitness function** — The objective function you want to minimize. Enter the fitness function in the form @fitnessfun, where fitnessfun.m is an M-file that computes the fitness function. "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.

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

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

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

For more information,

- See the "Optimization Tool" chapter in the *Optimization Toolbox™ User's Guide*.
- See "Example: Rastrigin's Function" on page 3-8 for an example of using the tool.

Example: Rastrigin's Function

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.

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

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

Genetic Algorithm and Direct Search Toolbox™ software contains an M-file, rastriginsfcn.m, that computes the values of Rastrigin's function. The following figure shows a plot of Rastrigin's function.

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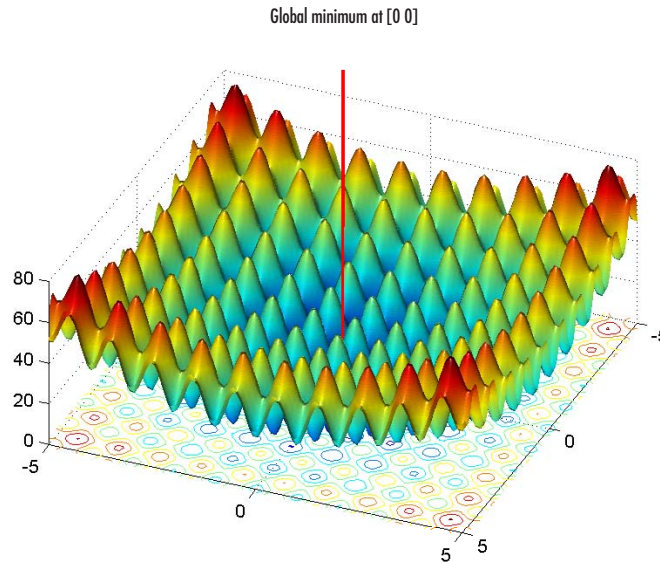
As the plot shows, Rastrigin's function has many local minima—the "valleys" in the plot. However, the function has just one global minimum, which occurs at the point [0 0] in the *x-y* plane, as indicated by the vertical line in the plot, where the value of the function is 0. At any local minimum other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

Local maxima

Local minima

Global minimum at [0 0]
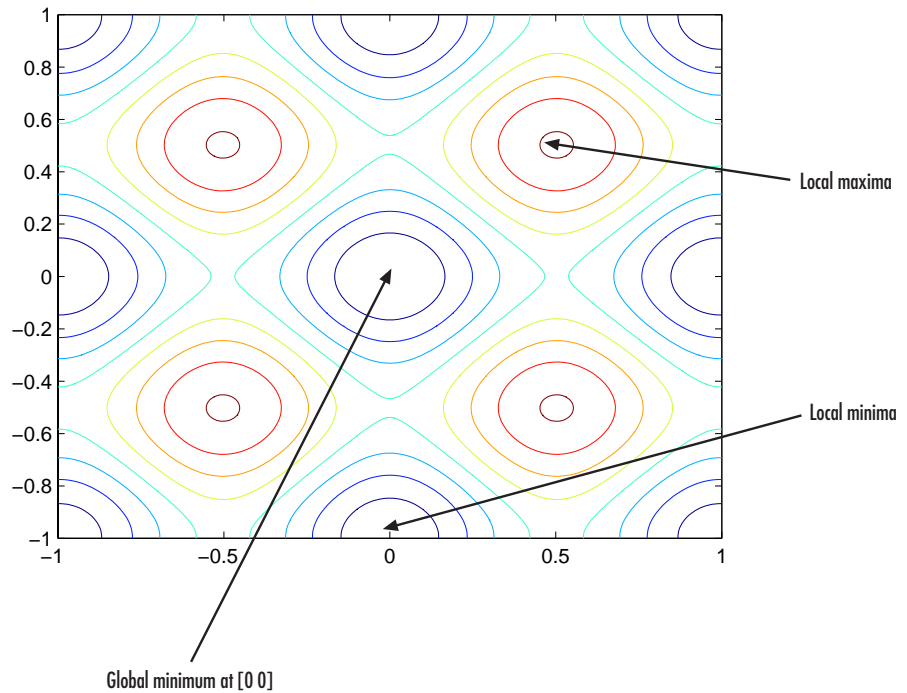
Finding the Minimum of Rastrigin's Function

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

Note Because the genetic algorithm uses random number generators, the algorithm returns slightly different results each time you run it.

To find the minimum, do the following steps:

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

2 Enter the following in the Optimization Tool:
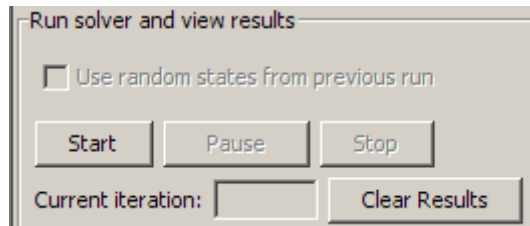
- In the **Fitness function** field, enter @rastriginsfcn.

- 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.

Fitness function:	@rastriginsfcn
Number of variables:	2

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

While the algorithm is running, the **Current iteration** 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 **Run solver and view results** pane appears as shown in the following figure.

The **Run solver and view results** pane displays the following information:
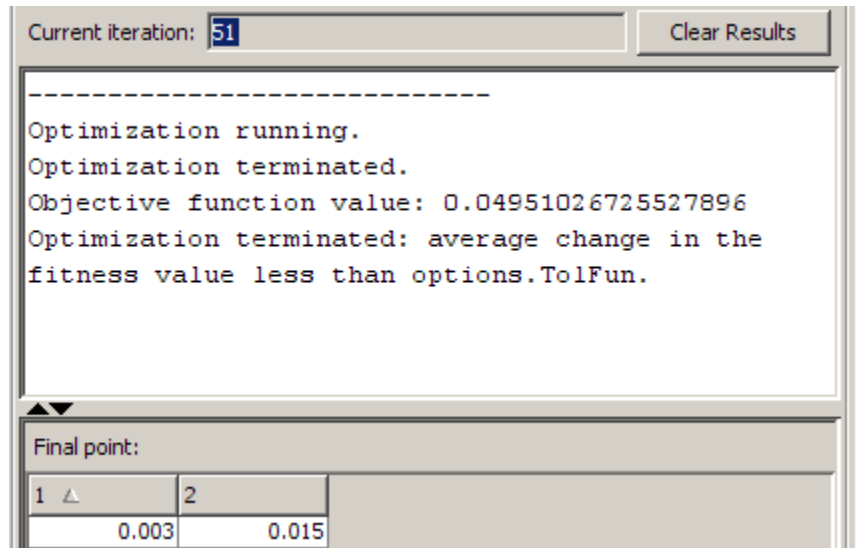
- The final value of the fitness function when the algorithm terminated:

    ```
    Objective function value: 0.04951026725527896
    ```

    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 6-22 describes some ways to get a result that is closer to the actual minimum.

- The reason the algorithm terminated.

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

- The final point, which in this example is [0.003 0.015].

Finding the Minimum from the Command Line

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

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

This returns

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

x =

    0.0229    0.0106


fval =

    0.1258


exitflag =

1
```

where

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

Note Because the genetic algorithm uses random number generators, the algorithm returns slightly different results each time you run it.

Displaying Plots

The **Plot functions** pane enables you to display various plots that provide information about the genetic algorithm while it is running. This information can help you change options to improve the performance of the algorithm. For example, to plot the best and mean values of the fitness function at each generation, select the box next to **Best fitness**, as shown in the following figure.

When you click **Start**, the Optimization Tool displays a plot of the best and mean values of the fitness function at each generation. When the algorithm stops, the plot appears as shown in the following figure.
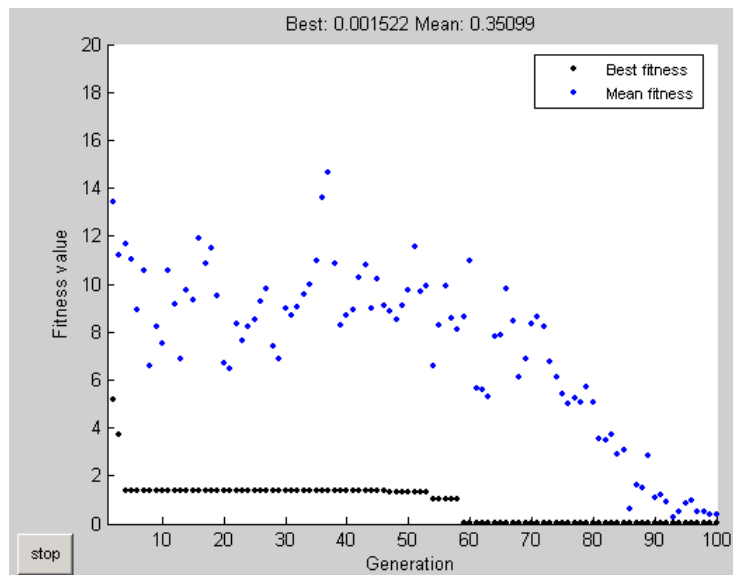
The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The plot also displays the best and mean values in the current generation numerically at the top.

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

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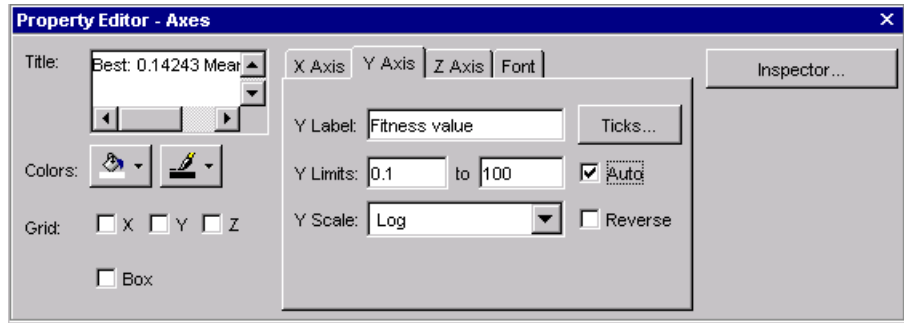
2 Click the **Y Axis** tab.

3 In the **Y Scale** pane, select **Log**.

The plot now appears as shown in the following figure.

Best: 0.0067796 Mean: 0.014788
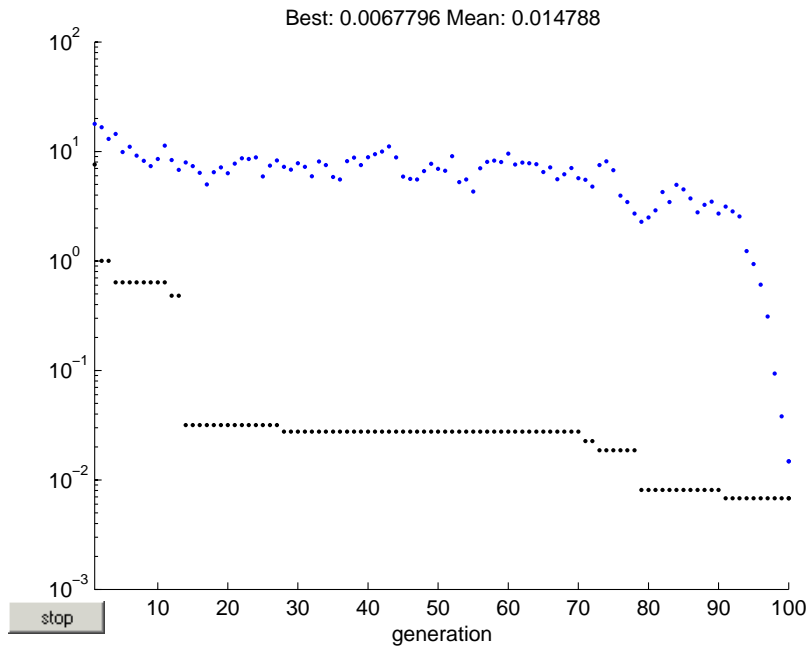
Typically, the best fitness value improves rapidly in the early generations, when the individuals are farther from the optimum. The best fitness value improves more slowly in later generations, whose populations are closer to the optimal point.

Some Genetic Algorithm Terminology

In this section...
"Fitness Functions" on page 3-17
"Individuals" on page 3-17
"Populations and Generations" on page 3-18
"Diversity" on page 3-18
"Fitness Values and Best Fitness Values" on page 3-18
"Parents and Children" on page 3-19

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 software tries to find the minimum of the fitness function.

You can write the fitness function as an M-file and pass it as a function handle 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.
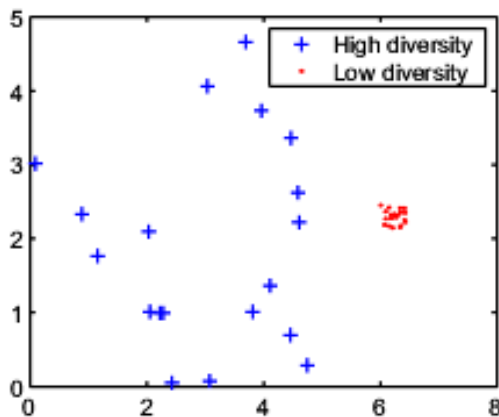
Diversity is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

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 software 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

In this section...
"Outline of the Algorithm" on page 3-20
"Initial Population" on page 3-21
"Creating the Next Generation" on page 3-22
"Plots of Later Generations" on page 3-24
"Stopping Conditions for the Algorithm" on page 3-24

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. At each step, the algorithm uses the individuals in the current generation to create the next population. To create the new population, 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 members, called parents, based on their fitness.

   d Some of the individuals in the current population that have lower fitness are chosen as *elite*. These elite individuals are passed to the next population.

   e 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*.

   f 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 3-24.

Initial Population

The algorithm begins by creating a random initial population, as shown in the following figure.
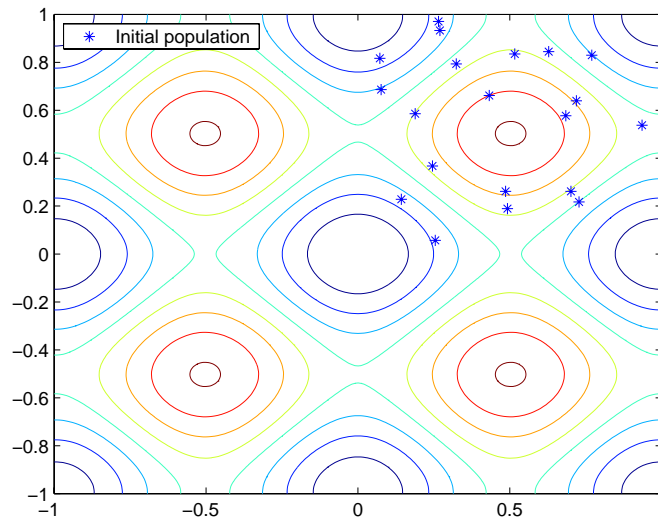
In this example, the initial population contains 20 individuals, which is the default value of **Population size** in the **Population** options. Note that all the individuals in the initial population lie in the upper-right quadrant of the picture, that is, their coordinates lie between 0 and 1, because 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.

Elite child

Crossover child
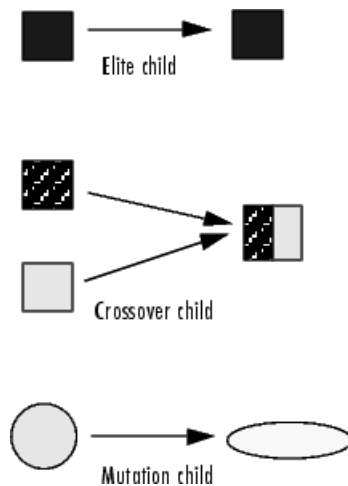
Mutation child

"Mutation and Crossover" on page 6-35 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.
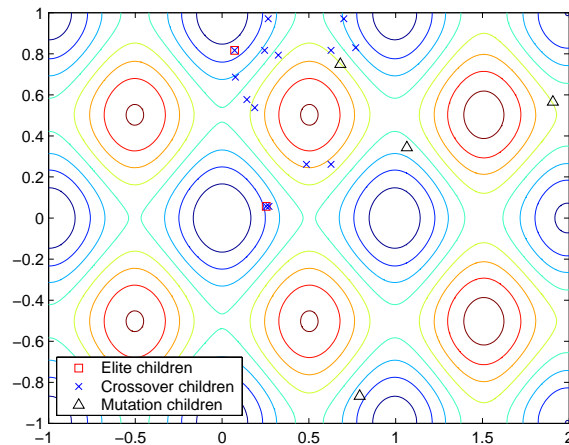
Plots of Later Generations

The following figure shows the populations at iterations 60, 80, 95, and 100.
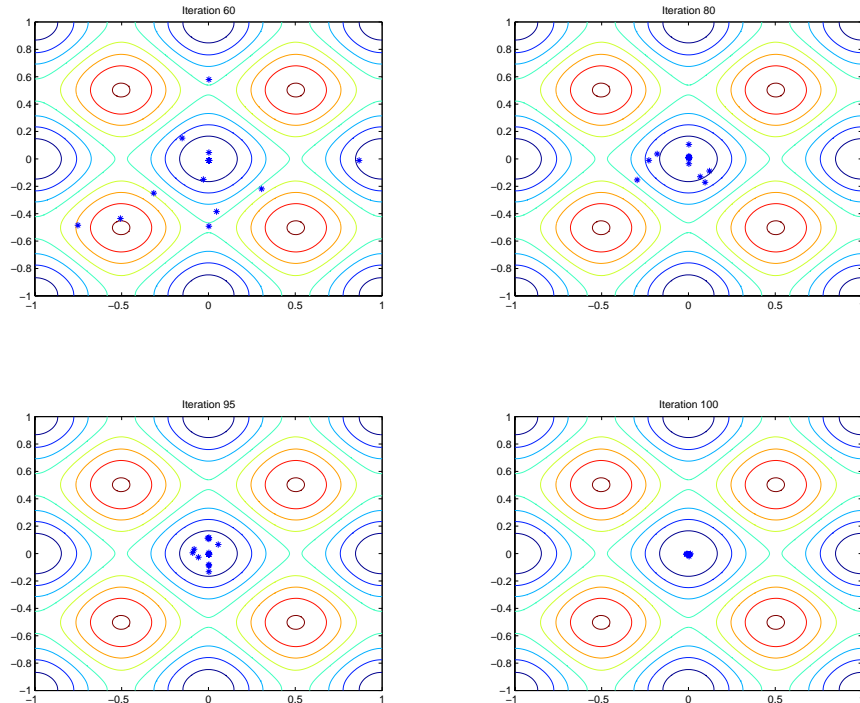
As the number of generations increases, the individuals in the population get closer together and approach the minimum point [0 0].

Stopping Conditions for the Algorithm

The genetic algorithm uses the following 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 when the weighted average change in the fitness function value over **Stall generations** is less than **Function tolerance**.

- **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**.

- **Function Tolerance** — The algorithm runs until the weighted average change in the fitness function value over **Stall generations** is less than **Function tolerance**.

- **Nonlinear constraint tolerance** — The **Nonlinear constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints.

The algorithm stops as soon as any one of these conditions is met. You can specify the values of these criteria in the **Stopping criteria** pane in the Optimization Tool. The default values are shown in the pane.

When you run the genetic algorithm, the **Run solver and view results** 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**.

Description of the Nonlinear Constraint Solver

The genetic algorithm uses the Augmented Lagrangian Genetic Algorithm (ALGA) to solve nonlinear constraint problems. The optimization problem solved by the ALGA algorithm is

$$\min_x f(x)$$

such that

$$
\begin{aligned}
c_i(x) &\leq 0,\ i = 1 \ldots m \\
ceq_i(x) &= 0,\ i = m+1 \ldots mt \\
A \cdot x &\leq b \\
Aeq \cdot x &= beq \\
lb \leq x &\leq ub,
\end{aligned}
$$

where $c(x)$ represents the nonlinear inequality constraints, $ceq(x)$ represents the equality constraints, m is the number of nonlinear inequality constraints, and mt is the total number of nonlinear constraints.

The Augmented Lagrangian Genetic Algorithm (ALGA) attempts to solve a nonlinear optimization problem with nonlinear constraints, linear constraints, and bounds. In this approach, bounds and linear constraints are handled separately from nonlinear constraints. A subproblem is formulated by combining the fitness function and nonlinear constraint function using the Lagrangian and the penalty parameters. A sequence of such optimization problems are approximately minimized using the genetic algorithm such that the linear constraints and bounds are satisfied.

A subproblem formulation is defined as

$$\Theta(x, \lambda, s, \rho) = f(x) - \sum_{i=1}^{m} \lambda_i s_i \log(s_i - c_i(x)) + \sum_{i=m+1}^{mt} \lambda_i c_i(x) + \frac{\rho}{2} \sum_{i=m+1}^{mt} c_i(x)^2,$$

where the components λ_i of the vector λ are nonnegative and are known as Lagrange multiplier estimates. The elements s_i of the vector s are nonnegative

shifts, and ρ is the positive penalty parameter. The algorithm begins by using an initial value for the penalty parameter (`InitialPenalty`).

The genetic algorithm minimizes a sequence of the subproblem, which is an approximation of the original problem. When the subproblem is minimized to a required accuracy and satisfies feasibility conditions, the Lagrangian estimates are updated. Otherwise, the penalty parameter is increased by a penalty factor (`PenaltyFactor`). This results in a new subproblem formulation and minimization problem. These steps are repeated until the stopping criteria are met. For a complete description of the algorithm, see the following references:

- Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds," *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

- Conn, A. R., N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds," *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

4

Getting Started with Simulated Annealing and Threshold Acceptance

What Are Simulated Annealing and Threshold Acceptance?

Simulated annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy.

At each iteration of the simulated annealing algorithm, a new point is randomly generated. The distance of the new point from the current point, or the extent of the search, is based on a probability distribution with a scale proportional to the temperature. The algorithm accepts all new points that lower the objective, but also, with a certain probability, points that raise the objective. By accepting points that raise the objective, the algorithm avoids being trapped in local minima, and is able to explore globally for more possible solutions. An *annealing schedule* is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum.

Threshold acceptance uses a similar approach, but instead of accepting new points that raise the objective with a certain probability, it accepts all new points below a fixed threshold. The threshold is then systematically lowered, just as the temperature is lowered in an annealing schedule. Because threshold acceptance avoids the probabilistic acceptance calculations of simulated annealing, it may locate an optimizer faster than simulated annealing.

Performing a Simulated Annealing or Threshold Acceptance Optimization

Calling simulannealbnd and threshacceptbnd at the Command Line

This section provides a brief introduction to using the simulated annealing and threshold acceptance at the command line. To simplify the presentation, this section uses only the simulated annealing function `simulannealbnd`. The threshold acceptance function `threshacceptbnd` has the same syntax.

To call the simulated annealing function at the command line, use the syntax

```
[x fval] = simulannealbnd(@objfun,x0,lb,ub,options)
```

where

- `@objfun` is a function handle to the objective function.
- `x0` is an initial guess for the optimizer.
- `lb` and `ub` are lower and upper bound constraints, respectively, on `x`.
- `options` is a structure containing options for the algorithm. If you do not pass in this argument, `simulannealbnd` uses its default options.

The results are given by:

- `x` — Final point returned by the solver
- `fval` — Value of the objective function at `x`

The command-line function `simulannealbnd` is convenient if you want to

- Return results directly to the MATLAB® workspace.
- Run the simulated annealing algorithm multiple times with different options by calling `simulannealbnd` from an M-file.

"Using the Simulated Annealing and Threshold Acceptance Algorithms from the Command Line" on page 7-2 provides a detailed description of using the function `simulannealbnd` and creating the options structure.

Using the Optimization Tool

This section provides a brief introduction to using the simulated annealing and threshold acceptance using the Optimization Tool. To simplify the presentation, this section uses only the simulated annealing function `simulannealbnd`. The threshold acceptance function `threshacceptbnd` has the same syntax.

To open the Optimization Tool, enter

```
optimtool('simulannealbnd')
```

at the command line, or enter `optimtool` and then choose `simulannealbnd` from the **Solver** menu.
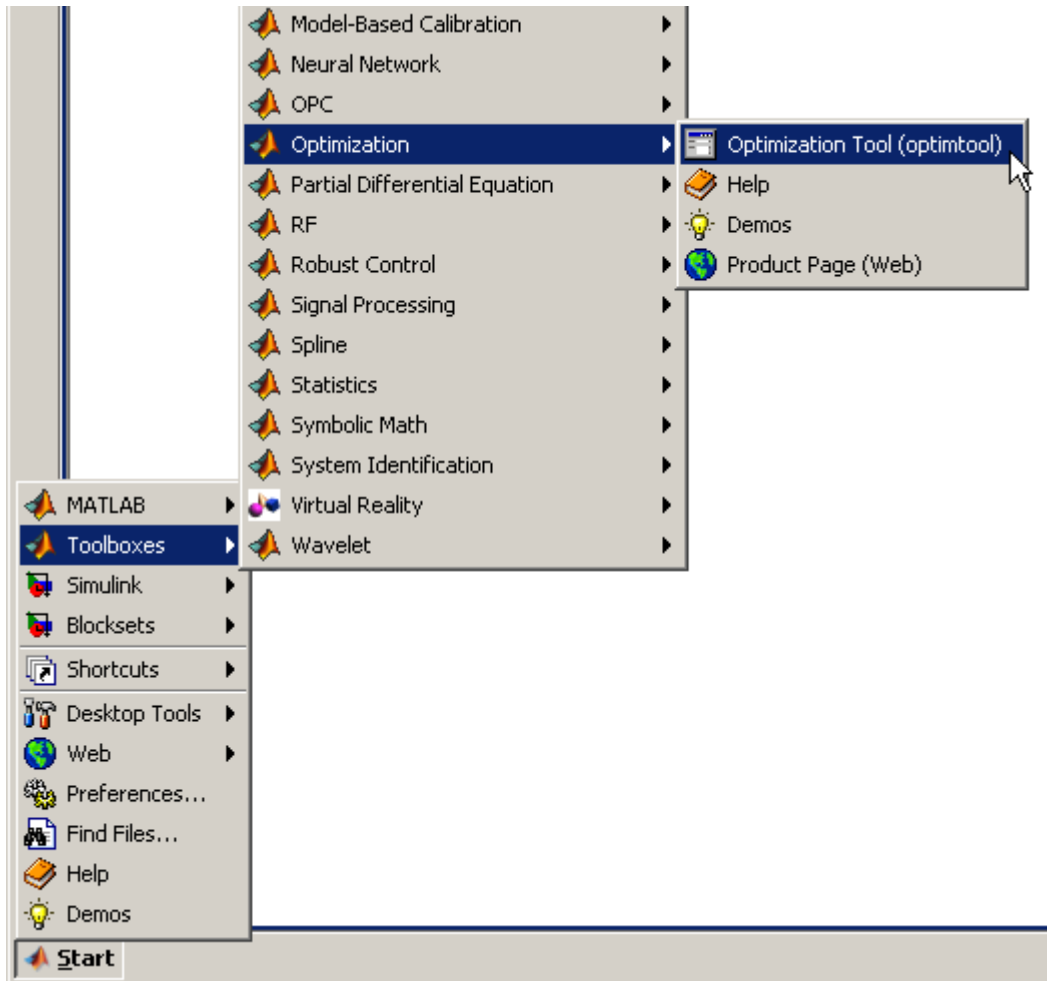
Set options

Expand or contract help

Choose solver

Enter problem
and constraints

Run solver

View results

See final point

You can also launch the tool from the MATLAB **Start** menu as pictured:

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

- **Objective 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 objective 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.

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

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

You can change the options for the simulated annealing 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 the "Optimization Tool" chapter in the *Optimization Toolbox™ User's Guide*.
- See "Minimizing Using the Optimization Tool" on page 4-9 for an example of using the tool with the function simulannealbnd.

Example: Minimizing De Jong's Fifth Function

In this section...
"Description" on page 4-8
"Minimizing at the Command Line" on page 4-9
"Minimizing Using the Optimization Tool" on page 4-9

Description

This section presents an example that shows how to find the minimum of the function using simulated annealing.

De Jong's fifth function is a two-dimensional function with many (25) local minima:

dejong5fcn
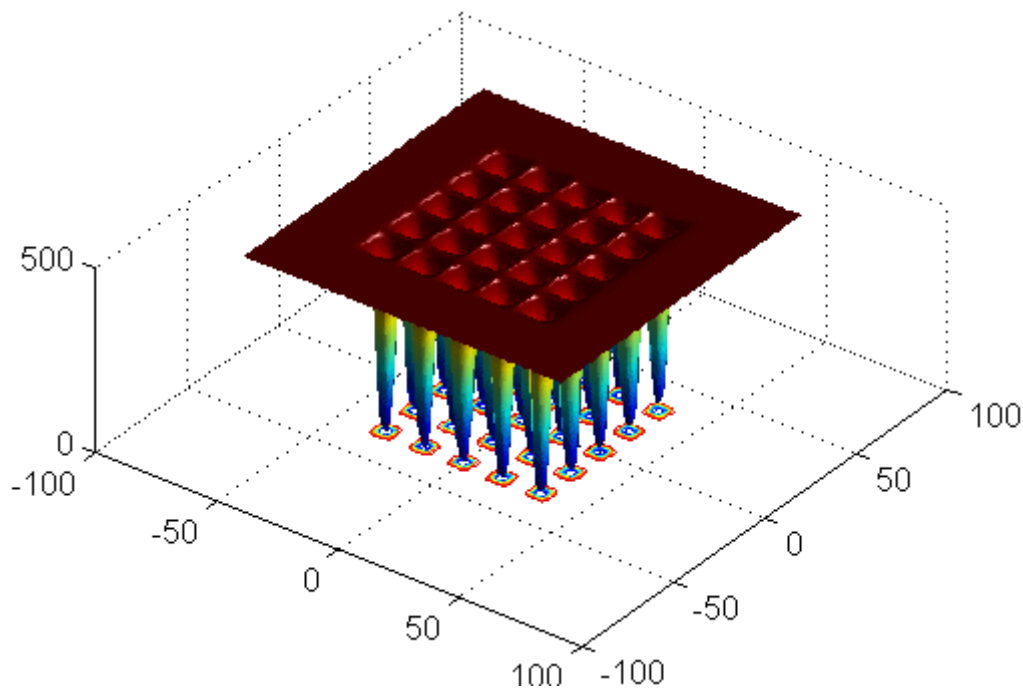
Many standard optimization algorithms get stuck in local minima. Because the simulated annealing algorithm performs a wide random search, the chance of being trapped in local minima is decreased.

Note Because simulated annealing and threshold acceptance both use random number generators, each time you run these algorithms you may get different results. See "Reproducing Your Results" on page 7-5 for more information.

Minimizing at the Command Line

To run the simulated annealing algorithm without constraints, call `simulannealbnd` at the command line using the objective function in `dejong5fcn.m`, referenced by anonymous function pointer:

```
fun = @dejong5fcn;
[x fval] = simulannealbnd(fun, [0 0])
```

This returns

```
x =
   -31.9779  -31.9595
fval =
     0.9980
```

where

- `x` is the final point returned by the algorithm.

- `fval` is the objective function value at the final point.

Minimizing Using the Optimization Tool

To run the minimization using the Optimization Tool,

1 Set up your problem as pictured in the Optimization Tool

Problem Setup and Results

Solver: simulannealbnd - Simulated anne... ▼

Problem

Objective function: @dejong5fcn ▼

Start point: [0,0]

Constraints:

Bounds: Lower: [ ] Upper: [ ]
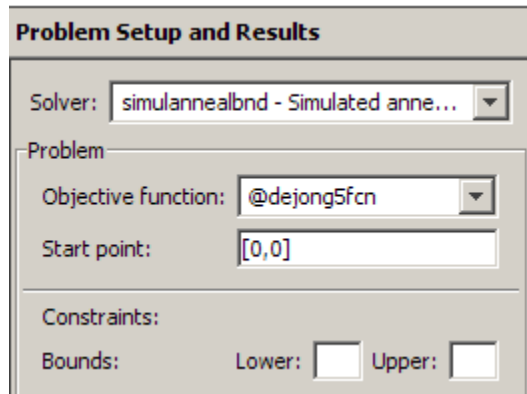
2 Click **Start** under **Run solver and view results**:

Run solver and view results

☐ Use random states from previous run

[Start] [Pause] [Stop]

Current iteration: 1853   [Clear Results]

```
----------------------------
Optimization running.
Optimization terminated.
Objective function value: 0.9980038386243747
Optimization terminated: change in best function
value less than options.TolFun.
```

Final point:

1 △	2
-31.991	-32.002
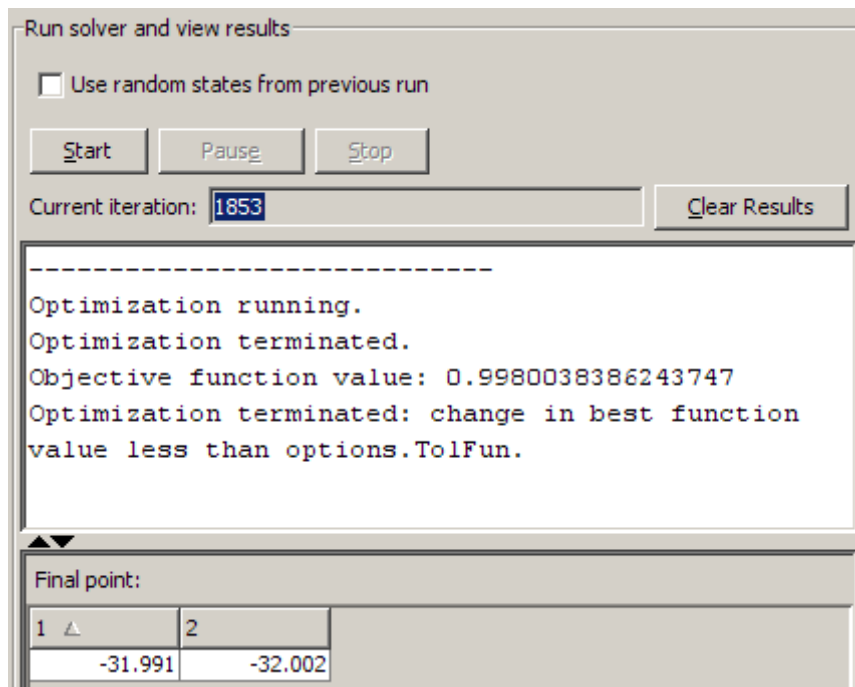
Some Simulated Annealing and Threshold Acceptance Terminology

In this section...
"Objective Function" on page 4-11
"Temperature" on page 4-11
"Annealing Schedule" on page 4-11
"Reannealing" on page 4-11

Objective Function

The *objective function* is the function you want to optimize. Genetic Algorithm and Direct Search Toolbox™ algorithms attempt to find the minimum of the objective function. Write the objective function as an M-file and pass it to the solver as afunction handle.

Temperature

The *temperature* is the control parameter in simulated annealing that is decreased gradually as the algorithm proceeds. It determines the probability of accepting a worse solution at any step and is used to limit the extent of the search in a given dimension. You can specify the initial temperature as an integer in the `InitialTemperature` option, and the annealing schedule as a function to the `TemperatureFcn` option.

Annealing Schedule

The *annealing schedule* is the rate by which the temperature is decreased as the algorithm proceeds. The slower the rate of decrease, the better the chances are of finding an optimal solution, but the longer the run time. You can specify the temperature schedule as a function handle with the `TemperatureFcn` option.

Reannealing

Annealing is the technique of closely controlling the temperature when cooling a material to ensure that it is brought to an optimal state. *Reannealing* raises

the temperature after a certain number of new points have been accepted, and starts the search again at the higher temperature. Reannealing avoids getting caught at local minima. You specify the reannealing schedule with the `ReannealInterval` option.

How Simulated Annealing and Threshold Acceptance Work

Outline of the Algorithm

The following is an outline of the steps performed for both the simulated annealing and threshold acceptance algorithms:

1 The algorithm begins by randomly generating a new point. The distance of the new point from the current point, or the extent of the search, is determined by a probability distribution with a scale proportional to the current temperature.

2 The algorithm determines whether the new point is better or worse than the current point. If the new point is better than the current point, it becomes the next point. If the new point is worse than the current point, the algorithm may still make it the next point. Simulated annealing accepts a worse point based on an acceptance probability. Threshold acceptance accepts a worse point if the objective function is raised by less than a fixed threshold.

3 The algorithm systematically lowers the temperature and (for threshold acceptance) the threshold, storing the best point found so far.

4 Reannealing is performed after a certain number of points (ReannealInterval) are accepted by the solver. Reannealing raises the temperature in each dimension, depending on sensitivity information. The search is resumed with the new temperature values.

5 The algorithm stops when the average change in the objective function is very small, or when any other stopping criteria are met. See "Stopping Conditions for the Algorithm" on page 4-14.

Stopping Conditions for the Algorithm

The simulated annealing and threshold acceptance algorithms use the following conditions to determine when to stop:

- `TolFun` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than `TolFun`. The default value is `1e-6`.

- `MaxIter` — The algorithm stops if the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. `Inf` is the default.

- `MaxFunEval` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the maximum number of function evaluations. The default maximum is `3000*numberofvariables`.

- `TimeLimit` specifies the maximum time in seconds the algorithm runs before stopping.

- `ObjectiveLimit` — The algorithm stops if the best objective function value is less than or equal to the value of `ObjectiveLimit`.

5

Using Direct Search

Performing a Pattern Search Using the Optimization Tool GUI

In this section...
"Example — A Linearly Constrained Problem" on page 5-2
"Displaying Plots" on page 5-5
"Example — Working with a Custom Plot Function" on page 5-6

Example — A Linearly Constrained Problem

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

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

where

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

$$f = [20 \quad 15 \quad 21 \quad 18 \quad 29 \quad 24],$$

subject to the constraints

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

where

$$A = [-8 \quad 7 \quad 3 \quad -4 \quad 9 \quad 0],$$
$$b = [7],$$
$$Aeq = \begin{bmatrix} 7 & 1 & 8 & 3 & 3 & 3 \\ 5 & 0 & 5 & 1 & 5 & 8 \\ 2 & 6 & 7 & 1 & 1 & 8 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$
$$beq = [84 \quad 62 \quad 65 \quad 1].$$

Performing a Pattern Search on the Example

To perform a pattern search on the example, first enter

```
optimtool('patternsearch')
```

to open the Optimization Tool, or enter `optimtool` and then choose `patternsearch` from the **Solver** menu. Then type the following function in the **Objective function** field:

```
@lincontest7
```

This is an M-file included in Genetic Algorithm and Direct Search Toolbox™ software 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 Optimization 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];
```
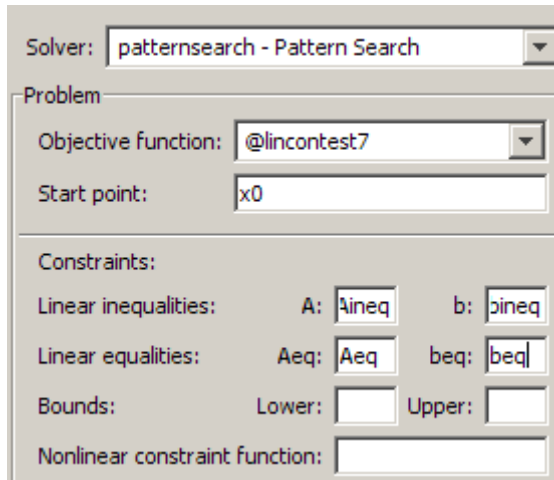
Then, enter the following in the Optimization Tool:

- Set **Start 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 Optimization Tool.

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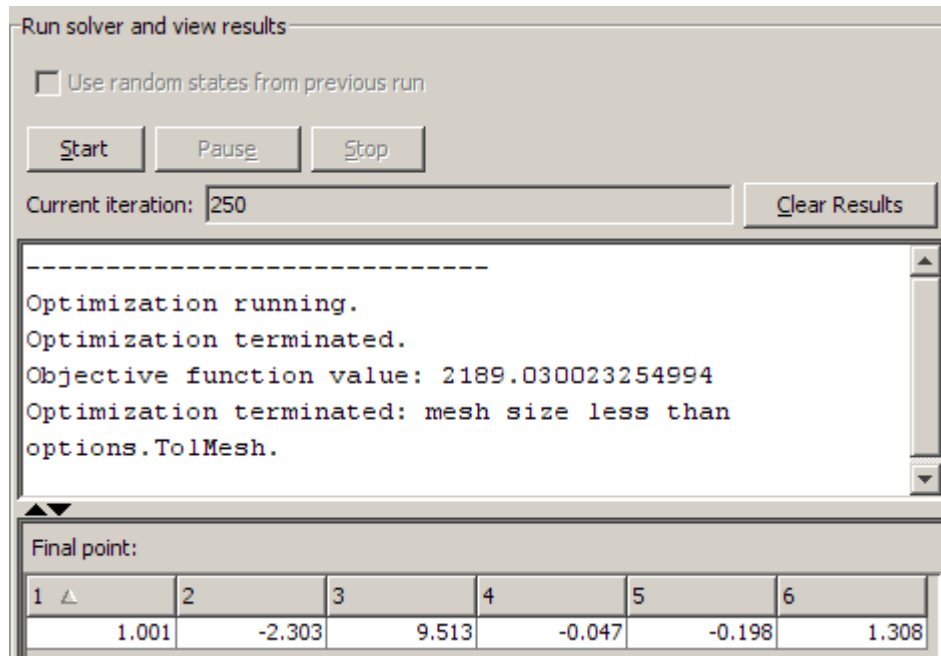
Displaying Plots

The **Plot functions** pane, shown in the following figure, enables you to display various plots of the results of a pattern search.
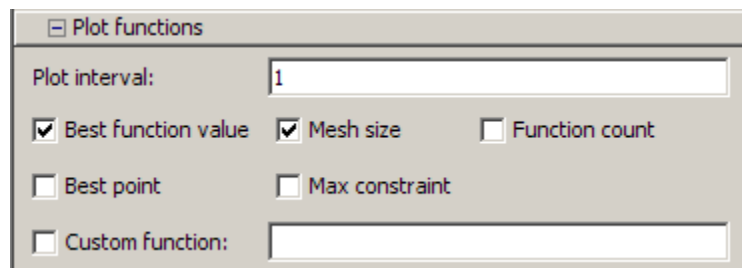
Select the check boxes next to the plots you want to display. For example, if you select **Best function value** and **Mesh size**, and run the example described in "Example: Finding the Minimum of a Function Using the GPS Algorithm" on page 2-7, the tool displays the plots shown in the following figure.
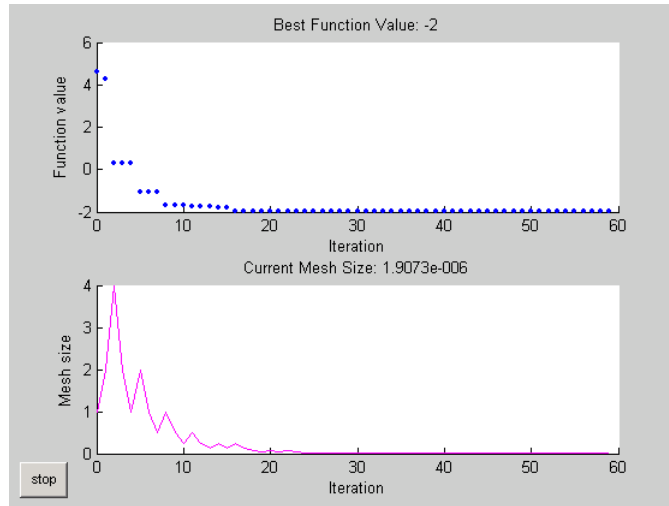
The upper plot displays the objective function value at each iteration. The lower plot displays the mesh size at each iteration.

Note When you display more than one plot, clicking on any plot while the pattern search is running or after the solver has completed opens a larger version of the plot in a separate window.

"Plot Options" on page 9-24 describes the types of plots you can create.

Example — Working with a Custom Plot Function

To use a plot function other than those included with the software, you can write your own custom plot function that is called at each iteration of the pattern search to create the plot. This example shows how to create a plot function that displays the logarithmic change in the best objective function value from the previous iteration to the current iteration.

This section covers the following topics:

- "Creating the Custom Plot Function" on page 5-7

- "Using the Custom Plot Function" on page 5-8

- "How the Plot Function Works" on page 5-9

Creating the Custom 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 stop = psplotchange(optimvalues, flag)
% PSPLOTCHANGE Plots the change in the best objective function
% value from the previous iteration.

% Best objective function value in the previous iteration
persistent last_best

stop = false;
if(strcmp(flag,'init'))
        set(gca,'Yscale','log'); %Set up the plot
        hold on;
        xlabel('Iteration');
        ylabel('Log Change in Values');
        title(['Change in Best Function Value']);
end

% Best objective function value in the current iteration
best = min(optimvalues.fval);

 % Set last_best to best
if optimvalues.iteration == O
last_best = best;

else
        %Change in objective function value
    change = last_best - best;
        plot(optimvalues.iteration, change, '.r');
end
```

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

Using the Custom Plot Function

To use the custom plot function, select **Custom function** in the **Plot functions** pane and enter @psplotchange in the field to the right. To compare the custom plot with the best function value plot, also select **Best function value**. Now, when you run the example described in "Example — A Linearly Constrained Problem" on page 5-2, the pattern search tool displays the plots shown in the following figure.
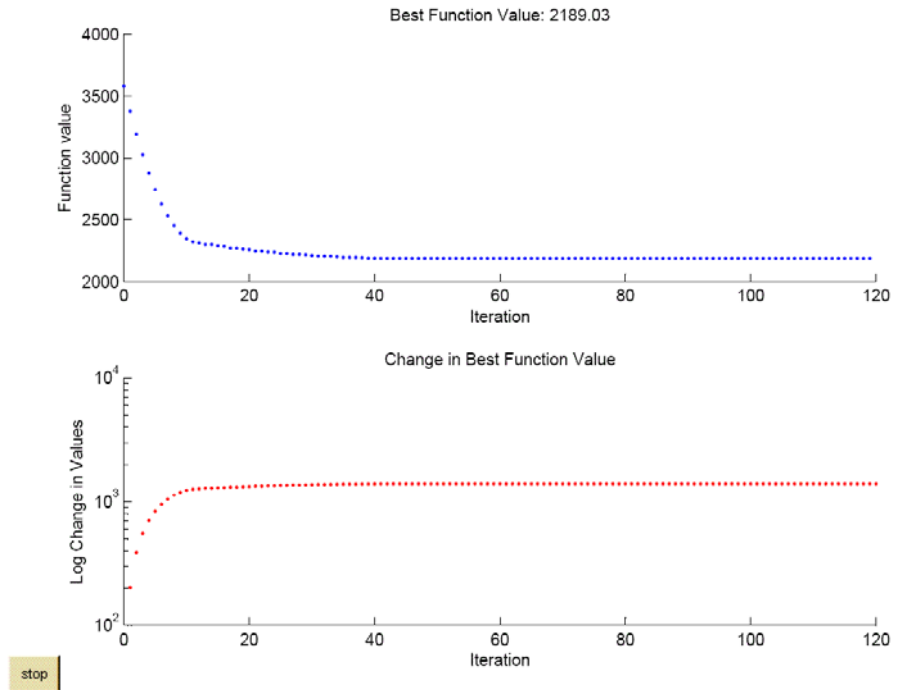
Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot will only show changes that are greater than 0. The logarithmic scale shows small changes in the objective function that the upper plot does not reveal.

How the Plot Function Works

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

- optimvalues — Structure containing the current state of the solver
- flag — String indicating the current status of the algorithm

The most important statements of the custom plot function, psplotchange.m, are summarized in the following table.

Custom Plot Function Statements

M-File Statement	Description
persistent last_best	Creates the persistent variable last_best, the best objective function value in the previous generation. Persistent variables are preserved over multiple calls to the plot function.
set(gca,'Yscale','log')	Sets up the plot before the algorithm starts.
best = min(optimvalues.fval)	Sets best equal to the minimum objective function value. The field optimvalues.fval contains the objective function value in the current iteration. The variable best is the minimum objective function value. For a complete description of the fields of the structure optimvalues, see "Structure of the Plot Functions" on page 9-4.

Custom Plot Function Statements (Continued)

M-File Statement	Description
`change = last_best - best`	Sets the variable change to the best objective function value at the previous iteration minus the best objective function value in the current iteration.
`plot(optimvalues.iteration, change, '.r')`	Plots the variable change at the current objective function value, for the current iteration contained in `optimvalues.iteration`.

Performing a Pattern Search from the Command Line

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 Using the GPS Algorithm" on page 2-7 from the command line by entering

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

This returns

```
Optimization terminated: mesh size less than options.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,nonlcon)
```

where

- A is a matrix and b is vector that represent inequality constraints of the form $A{\cdot}x \leq b$.

- Aeq is a matrix and beq is a vector that represent equality constraints of the form $Aeq{\cdot}x = beq$.

- lb and ub are vectors representing bound constraints of the form $lb \leq x$ and $x \leq ub$, respectively.

- nonlcon is a function that returns the nonlinear equality and inequality vectors, *c* and *ceq*, respectively. The function is minimized such that $c(x) \leq 0$ and $ceq(x) = 0$.

You only need to pass in the constraints that are part of the problem. For example, if there are no bound constraints or a nonlinear constraint function, 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 or a nonlinear constraint function, 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 available patternsearch options 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,nonlcon,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(@patternsearch)
```

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

```
options =
                TolMesh: 1.0000e-006
                 TolCon: 1.0000e-006
                   TolX: 1.0000e-006
                 TolFun: 1.0000e-006
```

```
             TolBind: 1.0000e-003
              MaxIter: '100*numberofvariables'
           MaxFunEvals: '2000*numberofvariables'
             TimeLimit: Inf
       MeshContraction: 0.5000
         MeshExpansion: 2
       MeshAccelerator: 'off'
            MeshRotate: 'on'
       InitialMeshSize: 1
             ScaleMesh: 'on'
           MaxMeshSize: Inf
        InitialPenalty: 10
         PenaltyFactor: 100
            PollMethod: 'gpspositivebasis2n'
          CompletePoll: 'off'
          PollingOrder: 'consecutive'
          SearchMethod: []
        CompleteSearch: 'off'
               Display: 'final'
            OutputFcns: []
              PlotFcns: []
          PlotInterval: 1
                 Cache: 'off'
             CacheSize: 10000
              CacheTol: 2.2204e-016
            Vectorized: 'off'
            UseParallel: 'never'
```

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.

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

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

Using Options and Problems from the Optimization Tool

As an alternative to creating the options structure using `psoptimset`, you can set the values of options in the Optimization Tool and then export the options to a structure in the MATLAB® workspace, as described in the "Importing and Exporting Your Work" section of the *Optimization Toolbox™ User's Guide*. If you export the default options in the Optimization 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 Optimization Tool.

You can also export an entire problem from the Optimization Tool and run it from the command line. Enter

```
patternsearch(problem)
```

where `problem` is the name of the exported problem.

Pattern Search Examples: Setting Options

In this section...

Note All examples use the generalized pattern search (GPS) algorithm, but can be applied to the MADS algorithm as well.

Poll Method

At each iteration, the 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 2-15 provides an example of polling. You can specify the pattern that defines the mesh by the **Poll method** option. The default pattern, GPS Positive basis 2N, consists of the following 2*N* directions, where *N* is the number of independent variables for the objective function.

[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[–1 0 0...0]
[0 –1 0...0]
[0 0 0...–1].

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

    [1 0 0]
    [0 1 0]
    [0 0 1]
    [–1 0 0]
    [0 –1 0]
    [0 0 –1].

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

    [1 0 0...0]
    [0 1 0...0]
    ...
    [0 0 0...1]
    [–1 –1 –1...–1].

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

    [1 0 0]
    [0 1 0]
    [0 0 1]
    [–1 –1 –1].

A pattern search will sometimes run faster using GPS Positive basis Np1 rather than the GPS Positive basis 2N as the **Poll method**, because the algorithm searches fewer points at each iteration. Although not being addressed in this example, the same is true when using the MADS Positive basis Np1 over the MADS Positive basis 2N. For example, if you run a pattern search on the example described in "Example — A Linearly Constrained Problem" on page 5-2, the algorithm performs 2080 function evaluations with GPS Positive basis 2N, the default **Poll method**, but only 1413 function evaluations using GPS Positive basis Np1.

However, if the objective function has many local minima, using GPS 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 Generalized 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.

Local minimum at (0,9)

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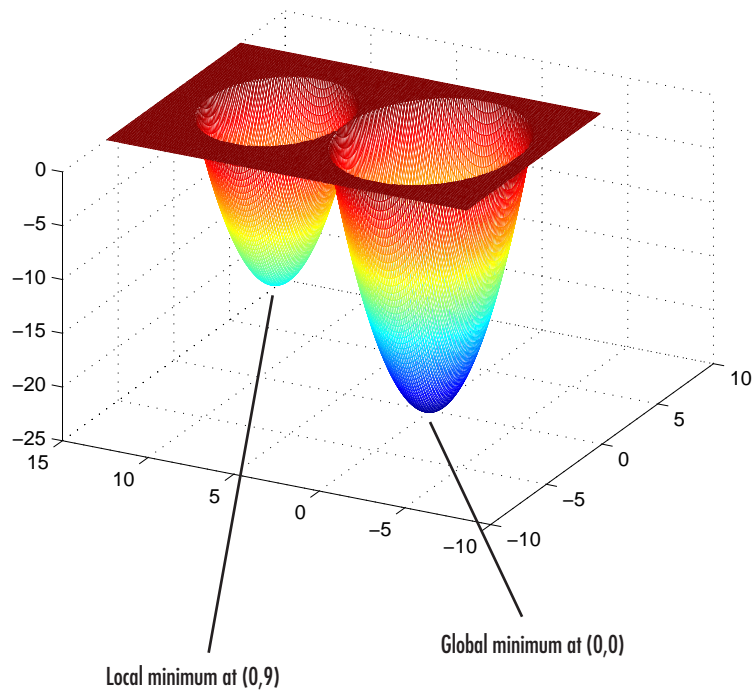
The global minimum of the function occurs at (0, 0), where its value is -25. However, the function also has a local minimum at (0, 9), where its value is -16.

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 Optimization Tool:

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

Click **Start** to run the pattern search with **Complete poll** set to `Off`, its default value. The Optimization Tool displays the results in the **Run solver and view results** pane, as shown in the following figure.
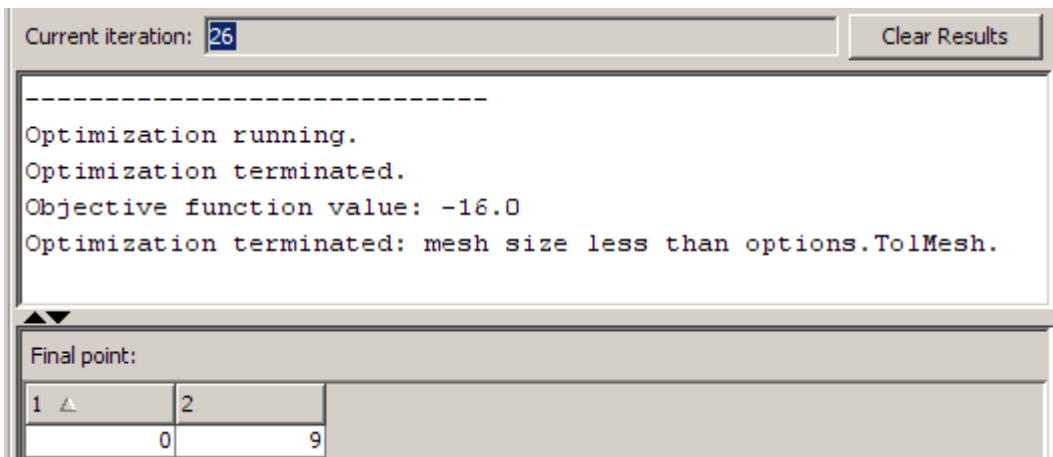
The pattern search returns the local minimum at (0, 9). At the initial point, (0, 5), the objective function value is 0. At the first iteration, the search polls the following mesh points.

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

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

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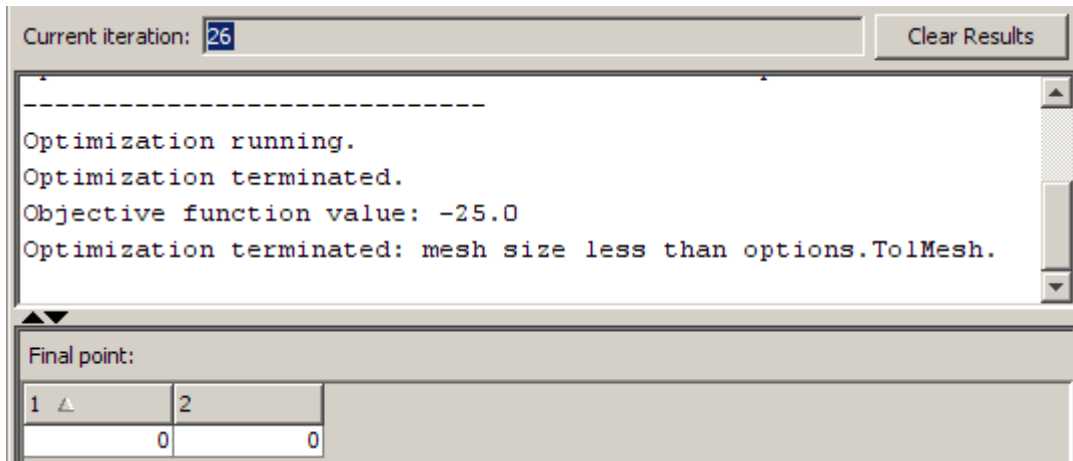
This time, the pattern search finds the global minimum at (0, 0). The difference between this run and the previous one is that with **Complete poll** set to On, at the first iteration the pattern search polls all four mesh points.

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

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

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

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

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.
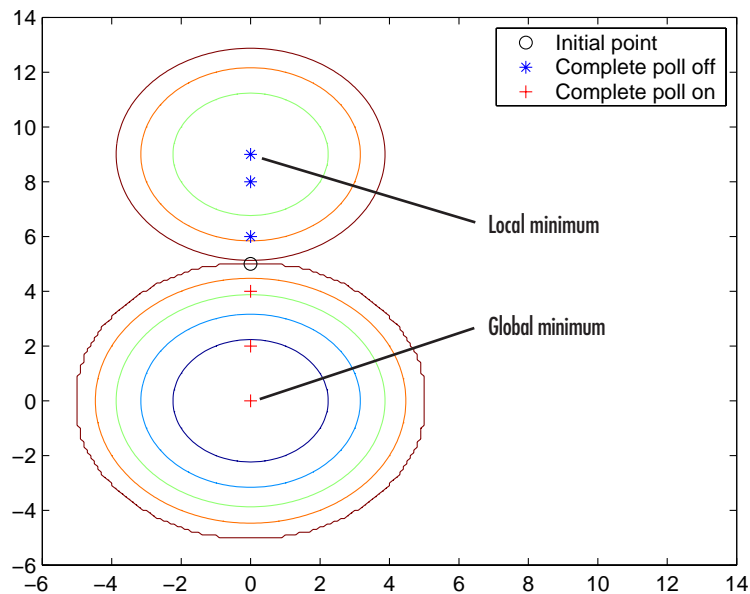
Using a Search Method

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 Linearly Constrained Problem" on page 5-2. 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 figures in the Optimization Tool.
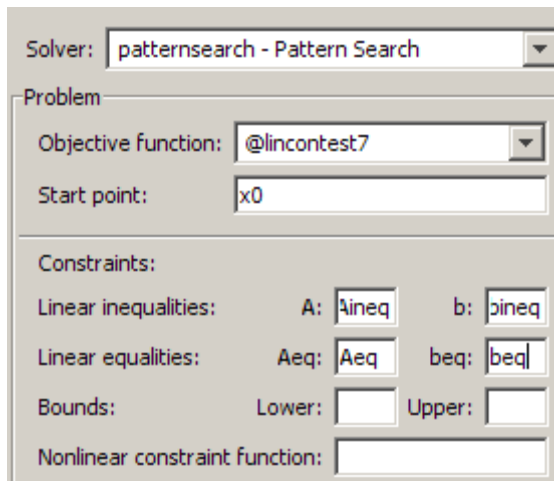
For comparison, click **Start** to run the example without a search method. This displays the plots shown in the following figure.
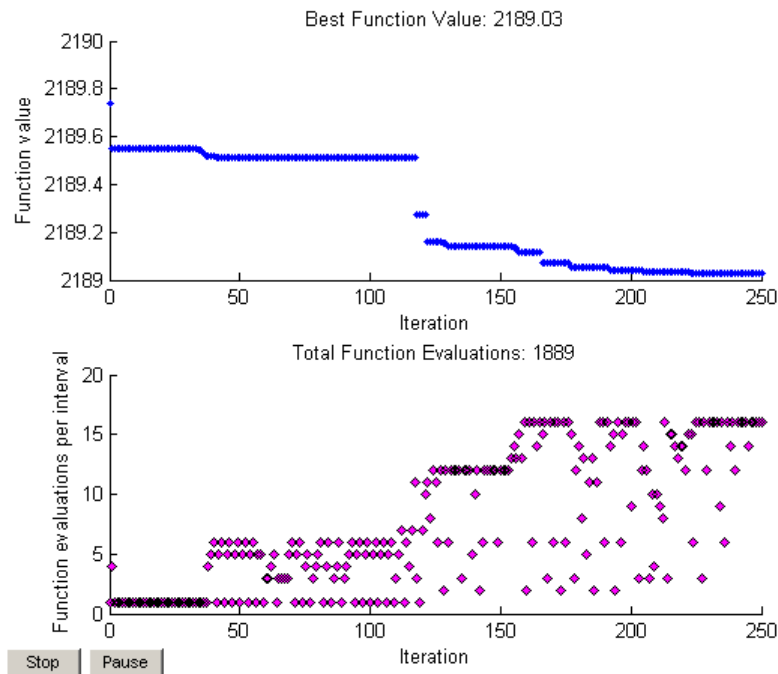
To see the effect of using a search method, select GPS Positive Basis Np1 in the **Search method** field in **Search** options. This sets the search method to be a pattern search using the pattern for GPS Positive basis Np1. Then click **Start** to run the genetic algorithm. This displays the following plots.
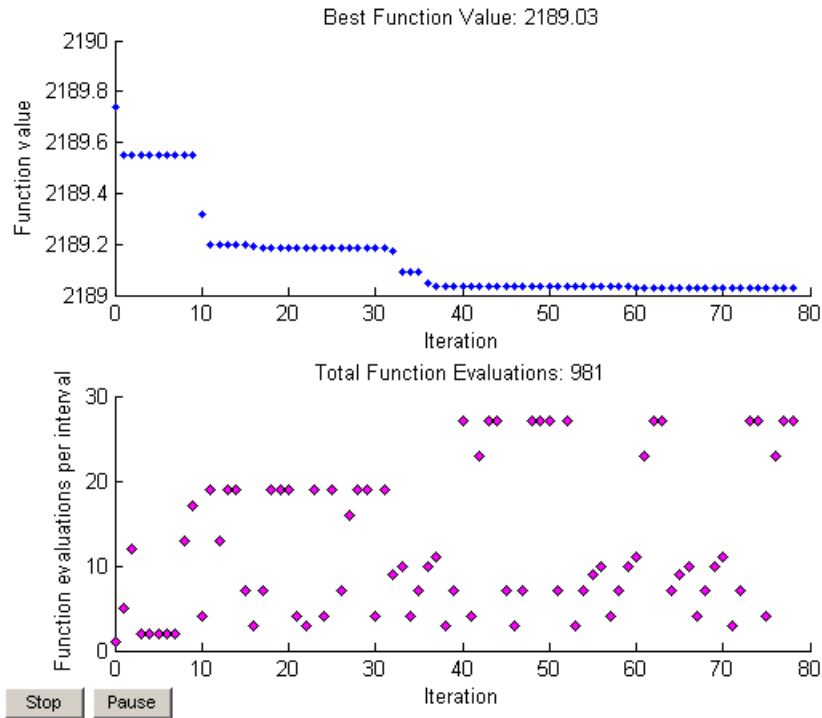
Note that using the search method reduces the total function evaluations by almost 50 percent—from 1889 to 981—and reduces the number of iterations from 250 to 78.

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 **Plot functions** pane.
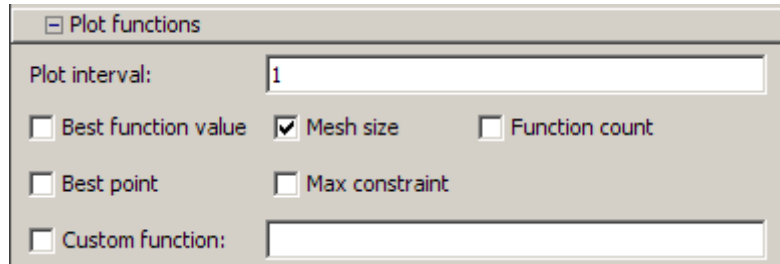
To also display the values of the mesh size and objective function at the command line, set **Level of display** to Iterative in the **Display to command window** options.
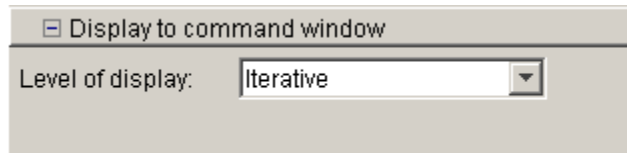
When you run the example described in "Example — A Linearly Constrained Problem" on page 5-2, the Optimization Tool displays the following plot.
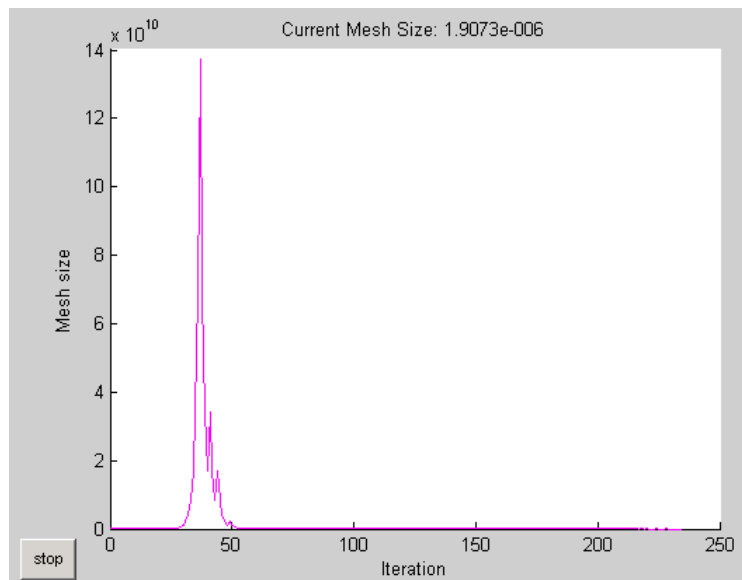
To see the changes in mesh size more clearly, change the *y*-axis to logarithmic scaling as follows:

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

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

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

Updating these settings in the MATLAB Property Editor will show the plot in the following figure.

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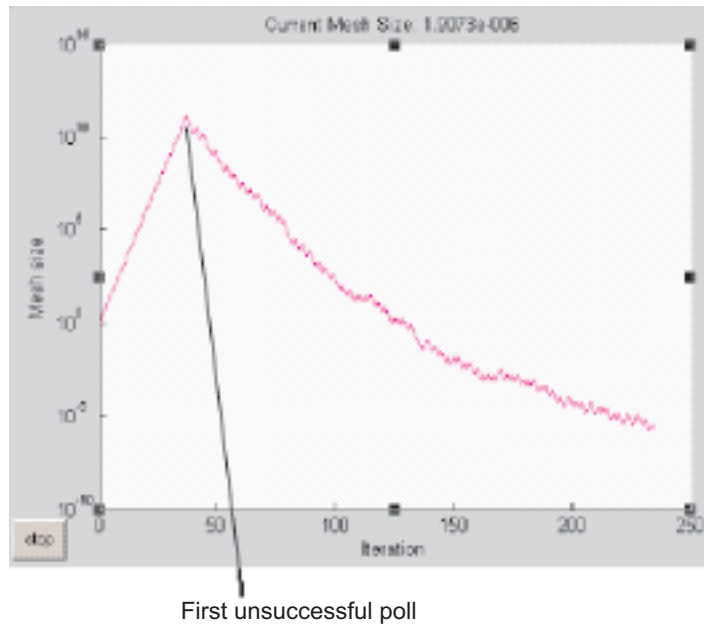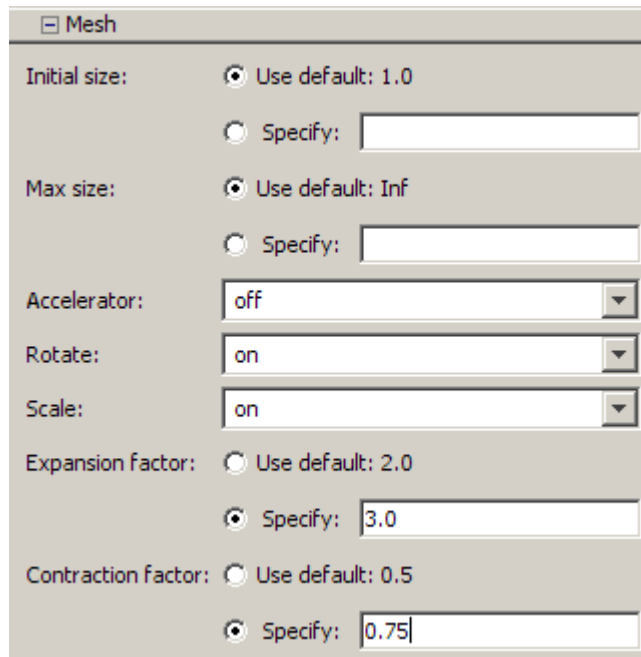
The first 37 iterations result in successful polls, so the mesh sizes increase steadily during this time. You can see that the first unsuccessful poll occurs at iteration 38 by looking at the command-line display for that iteration.

```
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 **Run solver and view 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.

Current iteration: |601| Clear Results

```
-----------------------------
Optimization running.
Optimization terminated.
Objective function value: 2189.030050449983
Maximum number of iterations exceeded: increase options.MaxIter.
```

Final point:

1 △	2	3	4	5	6
1.001	-2.303	9.513	-0.047	-0.198	1.308
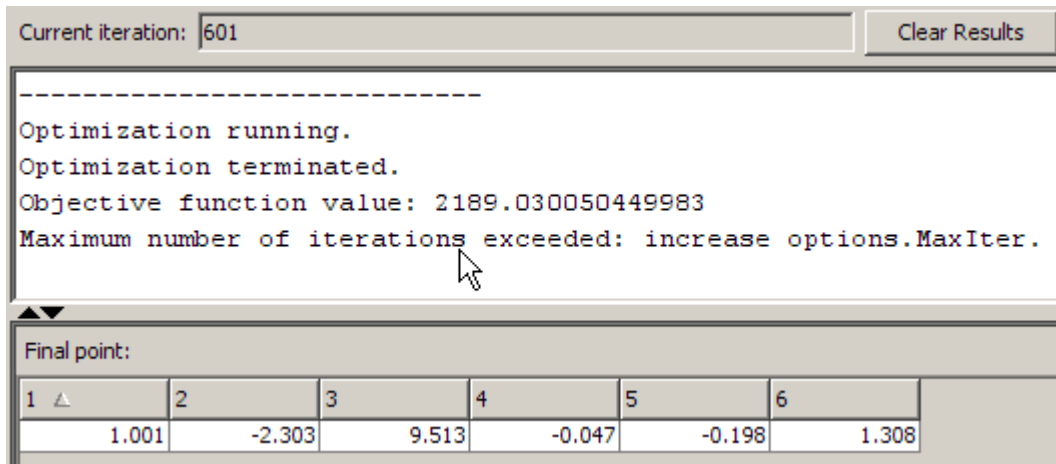
The algorithm halts because it exceeds the maximum number of iterations, whose value you can set in the **Max iteration** field in the **Stopping criteria** options. The default value is 100 times the number of variables for the objective function, which is 6 in this example.

When you change the scaling of the y-axis to logarithmic, the mesh size plot appears as shown in the following figure.
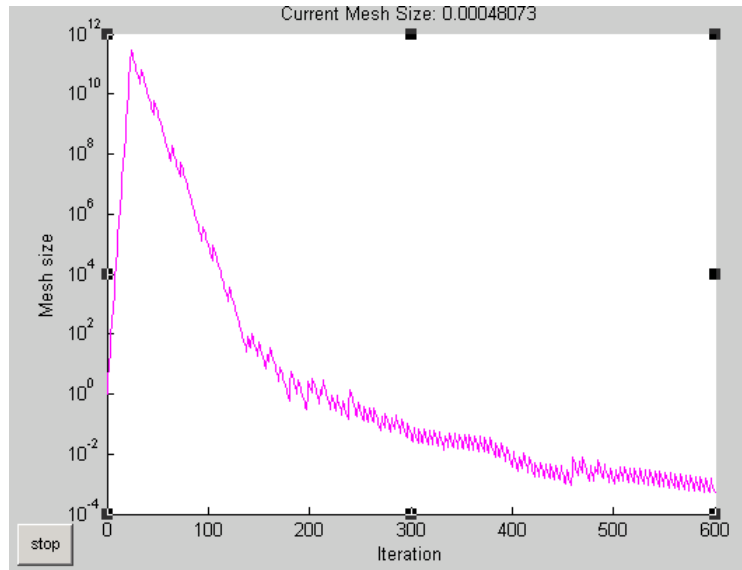
Note that the mesh size increases faster with **Expansion factor** set to 3.0, as compared with the default value of 2.0, and decreases more slowly with **Contraction factor** set to 0.75, as compared with the default value of 0.5.

Mesh Accelerator

The mesh accelerator can make a pattern search converge faster to an 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 It is recommended to 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 the **Mesh** options. When you run the example described in "Example — A Linearly Constrained

Problem" on page 5-2, 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 **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**.

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 0n, 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 0n than with **Cache** set to 0ff. 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 **Plot functions** pane and run the example described in "Example — A Linearly Constrained Problem" on page 5-2 with **Cache** set to 0ff. After the pattern search finishes, the plots appear as shown in the following figure.
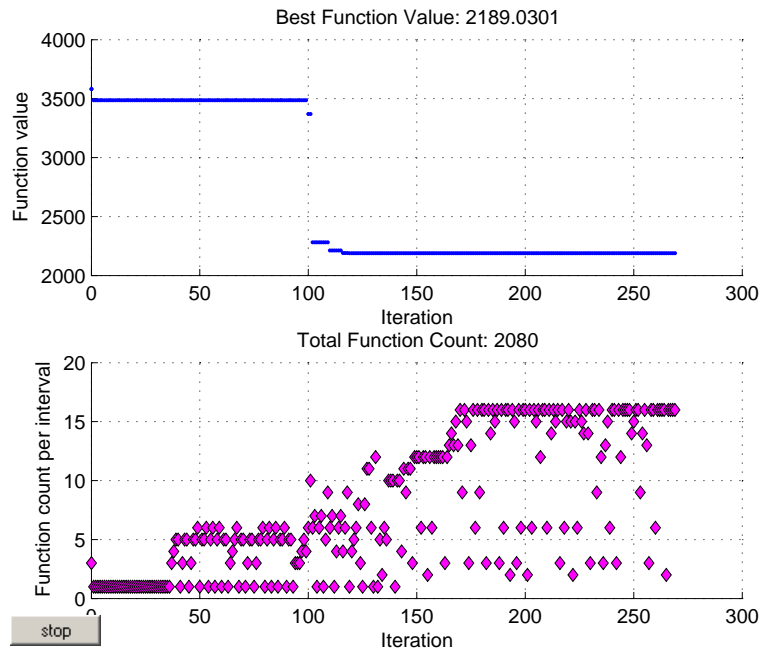
Note that the total function count is 2080.

Now, set **Cache** to 0n and run the example again. This time, the plots appear as shown in the following figure.
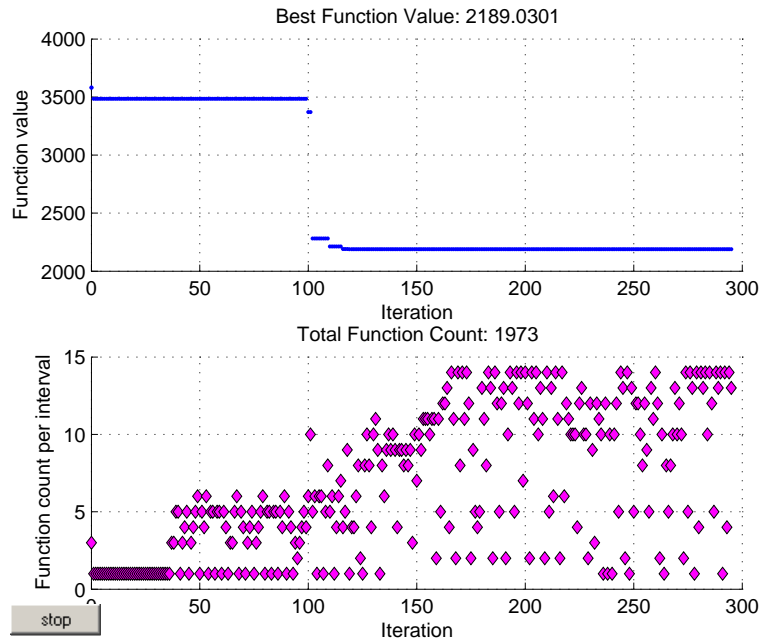
This time, the total function count is reduced to 1973.

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 difference between the function value at the previous best point and function value at the current best point is less than the value of **Function tolerance**, the algorithm halts.

- **Nonlinear constraint tolerance** — The algorithm treats a point to be feasible if constraint violation is less than TolCon.

- **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

$$-11x_1 + 10x_2 \leq 10$$
$$10x_1 - 10x_2 \leq 10.$$

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.

Feasible region
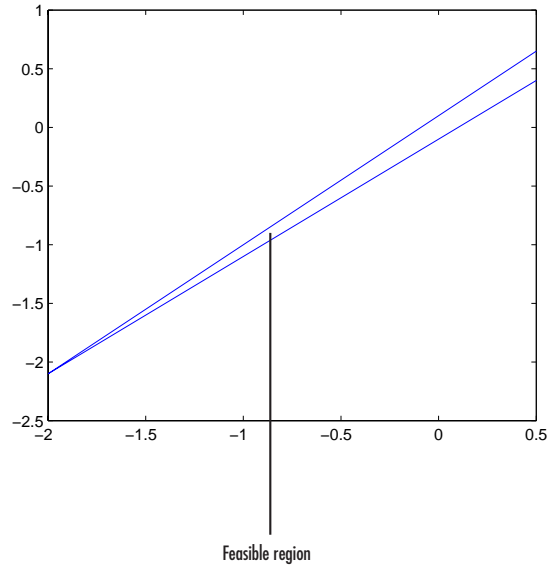
Running a Pattern Search with the Default Bind Tolerance

To run the example, enter `optimtool` and choose `patternsearch` to open the Optimization Tool. Enter the following information:

- Set **Objective function** to `@(x) norm(x)`.

- Set **Start point** to `[-1.001 -1.1]`.

- Select **Mesh size** in the `Plot functions` 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.

Initial Point and Mesh Points at Iteration 5
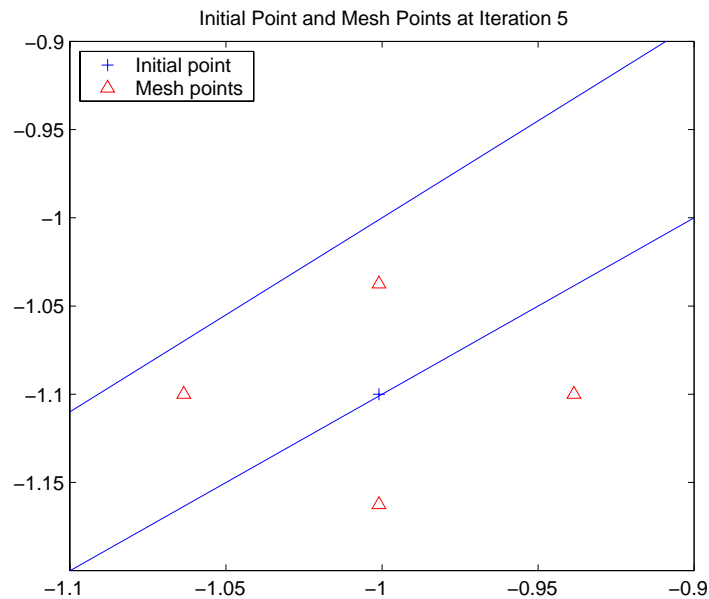
The top mesh point, which is (-1.001, -1.0375), has a smaller objective function value than the initial point, so the poll is successful.

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 $10x_1 - 10x_2 \leq 10$ 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 $10x_1 - 10x_2 \leq 10$, resulting in successful poll. The following figure shows the initial point with two addition search points in directions parallel to the boundary.
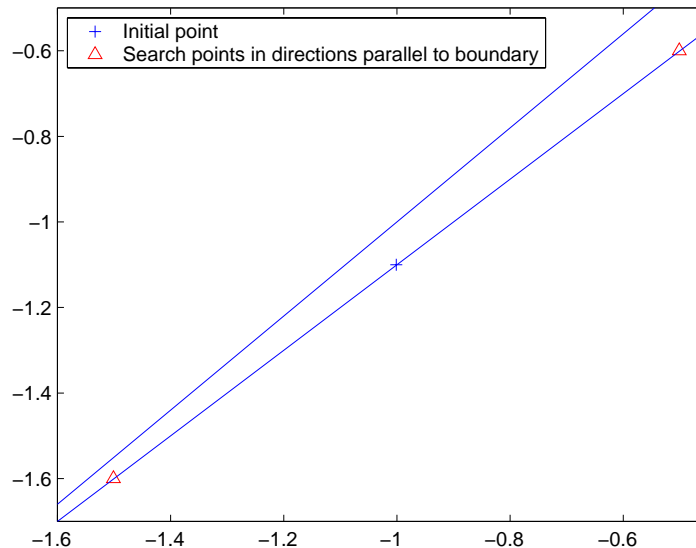
The following figure compares the sequences of points during the first 20 iterations of the pattern search for both settings of **Bind tolerance**.

First 20 Iterations for Two Settings of Bind Tolerance
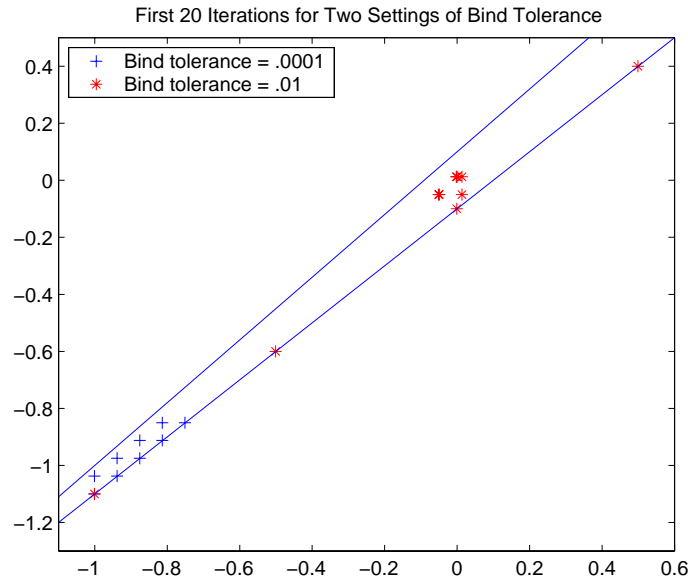
Note that when **Bind tolerance** is set to .01, the points move toward the optimal point more quickly. The pattern search requires only 90 iterations. When **Bind tolerance** is set to .0001, the search requires 124 iterations. However, when the feasible region does not contain very acute angles, as it does in this example, increasing **Bind tolerance** can increase the number of iterations required, because the pattern search tends to poll more points.

Constrained Minimization Using patternsearch

Suppose you want to minimize the simple objective function of two variables x1 and x2,

$$\min_{x} f(x) = \left(4 - 2.1x_1^2 - x_1^{4/3}\right)x_1^2 + x_1 x_2 + \left(-4 + 4x_2^2\right)x_2^2$$

subject to the following nonlinear inequality constraints and bounds

$$x_1 \cdot x_2 + x_1 - x_2 + 1.5 \le 0, \text{ (nonlinear constraint)}$$
$$10 - x_1 \cdot x_2 \le 0, \qquad \text{(nonlinear constraint)}$$
$$0 \le x_1 \le 1, \qquad \text{(bound)}$$
$$0 \le x_2 \le 13. \qquad \text{(bound)}$$

Begin by creating the objective and constraint functions. First, create an M-file named `simple_objective.m` as follows:

```
function y = simple_objective(x)
y = (4 - 2.1*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2) + (-4 + 4*x(2)^2)*x(2)^2;
```

The pattern search solver assumes the objective function will take one input x where x has as many elements as number of variables in the problem. The objective function computes the value of the function and returns that scalar value in its one return argument y.

Then create an M-file named `simple_constraint.m` containing the constraints:

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

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

Next, to minimize the objective function using the `patternsearch` function, you need to pass in a function handle to the objective function as well as specifying a start point as the second argument. Lower and upper bounds are provided as LB and UB respectively. In addition, you also need to pass a function handle to the nonlinear constraint function.

```
ObjectiveFunction = @simple_objective;
X0 = [0 0];   % Starting point
LB = [0 0];   % Lower bound
UB = [1 13];  % Upper bound
ConstraintFunction = @simple_constraint;
```

```
[x,fval] = patternsearch(ObjectiveFunction,X0,[],[],[],[],...
                         LB,UB,ConstraintFunction)

Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
    0.8122   12.3122

fval =
  9.1324e+004
```

Next, plot the results. Create an options structure using `psoptimset` that selects two plot functions. The first plot function `psplotbestf` plots the best objective function value at every iteration. The second plot function `psplotmaxconstr` plots the maximum constraint violation at every iteration.

Note You can also visualize the progress of the algorithm by displaying information to the Command Window using the `'Display'` option.

```
options = psoptimset('PlotFcns',{@psplotbestf,@psplotmaxconstr},'Display','iter');
[x,fval] = patternsearch(ObjectiveFunction,X0,[],[],[],[],LB,UB,ConstraintFunction,options)
```

			max		
Iter	f-count	f(x)	constraint	MeshSize	Method
0	1	0	10	0.8919	
1	5	113580	0	0.001	Increase penalty
2	24	91324.4	0	1e-005	Increase penalty
3	48	91324	0	1e-007	Increase penalty

```
Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
    0.8122   12.3122

fval =
  9.1324e+004
```
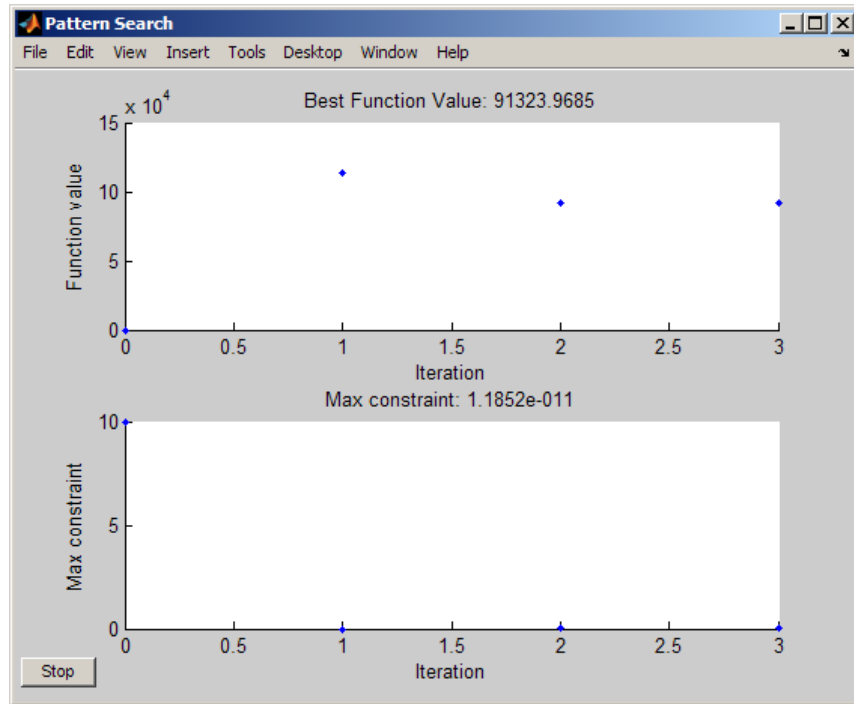
Best Objective Function Value and Maximum Constraint Violation at Each Iteration

Vectorizing the Objective and Constraint Functions

Direct search often runs faster if you *vectorize* the objective and nonlinear constraint functions. This means your functions evaluate all the points in a poll or search pattern at once, with one function call, without having to loop through the points one at a time. Therefore, the option Vectorize = 'on' works only when CompletePoll or CompleteSearch are also set to 'on'.

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

Vectorized Objective Function

A vectorized objective function accepts a matrix as input and generates a vector of function values, where each function value corresponds to one row or column of the input matrix. patternsearch resolves the ambiguity in whether the rows or columns of the matrix represent the points of a pattern as follows. Suppose the input matrix has m rows and n columns:

- If the initial point x0 is a column vector of size m, the objective function takes each column of the matrix as a point in the pattern and returns a vector of size n.

- If the initial point x0 is a row vector of size n, the objective function takes each row of the matrix as a point in the pattern and returns a vector of size m.

- If the initial point x0 is a scalar, the matrix has one row (m = 1, the matrix is a vector), and each entry of the matrix represents one point to evaluate.

Pictorially, the matrix and calculation are represented by the following figure.
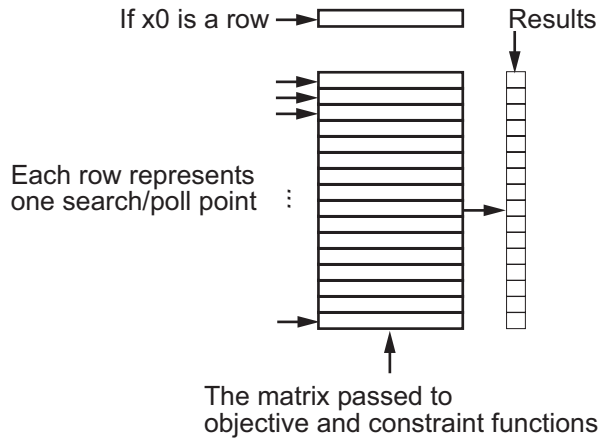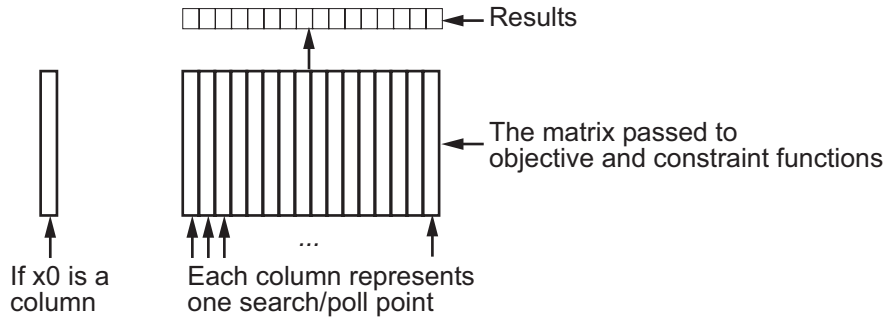
Structure of Vectorized Functions

For example, suppose the objective function is

$$f(x) = x_1^4 + x_2^4 - 4x_1^2 - 2x_2^2 + 3x_1 - x_2/2.$$

If the initial vector x0 is a column vector, such as [0;0], an M-file for vectorized evaluation is

```
function f = vectorizedc(x)

f = x(1,:).^4+x(2,:).^4-4*x(1,:).^2-2*x(2,:).^2...
    +3*x(1,:)-.5*x(2,:);
```

If the initial vector x0 is a row vector, such as [0,0], an M-file for vectorized evaluation is

```
function f = vectorizedr(x)

f = x(:,1).^4+x(:,2).^4-4*x(:,1).^2-2*x(:,2).^2...
    +3*x(:,1)-.5*x(:,2);
```

If you want to use the same objective (fitness) function for both pattern search and genetic algorithm, write your function to have the points represented by row vectors, and write x0 as a row vector. The genetic algorithm always takes individuals as the rows of a matrix. This was a design decision—the genetic algorithm does not require a user-supplied population, so needs to have a default format.

To minimize vectorizedc, enter the following commands:

```
options=psoptimset('Vectorized','on','CompletePoll','on');
x0=[0;0];
[x fval]=patternsearch(@vectorizedc,x0,...
        [],[],[],[],[],[],options)
```

MATLAB returns the following output:

```
Optimization terminated: mesh size less than options.TolMesh.

x =
   -1.5737
    1.0575

fval =
  -10.0088
```

Vectorized Constraint Functions

Only nonlinear constraints need to be vectorized; bounds and linear constraints are handled automatically. If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

The same considerations hold for constraint functions as for objective functions: the initial point x0 determines the type of points (row or column vectors) in the poll or search. If the initial point is a row vector of size k, the matrix x passed to the constraint function has k columns. Similarly, if the initial point is a column vector of size k, the matrix of poll or search points has k rows. The figure Structure of Vectorized Functions on page 5-44 may make this clear.

Your nonlinear constraint function returns two matrices, one for inequality constraints, and one for equality constraints. Suppose there are n_c nonlinear inequality constraints and n_{ceq} nonlinear equality constraints. For row vector x0, the constraint matrices have n_c and n_{ceq} columns respectively, and the number of rows is the same as in the input matrix. Similarly, for a column vector x0, the constraint matrices have n_c and n_{ceq} rows respectively, and the number of columns is the same as in the input matrix. In figure Structure of Vectorized Functions on page 5-44, "Results" includes both n_c and n_{ceq}.

Example of Vectorized Objective and Constraints

Suppose that the nonlinear constraints are

$$\frac{x_1^2}{9} + \frac{x_2^2}{4} \leq 1 \text{ (the interior of an ellipse)},$$
$$x_2 \geq \cosh(x_1) - 1.$$

Write an M-file for these constraints for row-form x0 as follows:

```
function [c ceq] = ellipsecosh(x)

c(:,1)=x(:,1).^2/9+x(:,2).^2/4-1;
c(:,2)=cosh(x(:,1))-x(:,2)-1;
ceq=[];
```

Minimize vectorizedr (defined in "Vectorized Objective Function" on page 5-43) subject to the constraints ellipsecosh:

```
x0=[0,0];
options=psoptimset('Vectorized','on','CompletePoll','on');
[x fval]=patternsearch(@vectorizedr,x0,...
        [],[],[],[],[],[],@ellipsecosh,options);
```

MATLAB returns the following output:

```
Optimization terminated: mesh size less than options.TolMesh
 and constraint violation is less than options.TolCon.

x =
   -1.3516    1.0612

fval =
   -9.5394
```

Parallel Computing with Pattern Search

In this section...
"Parallel Pattern Search" on page 5-48
"Using Parallel Computing with patternsearch" on page 5-49
"Parallel Search Function" on page 5-51
"Implementation Issues in Parallel Pattern Search" on page 5-51
"Parallel Computing Considerations" on page 5-52

Parallel Pattern Search

Parallel computing is the technique of using multiple processes or processors on a single problem. The reason to use parallel computing is to speed up computations.

If enabled for parallel computation, the Genetic Algorithm and Direct Search Toolbox™ solver patternsearch automatically distributes the evaluation of objective and constraint functions associated with the points in a pattern to multiple processes or processors. patternsearch uses parallel computing under the following conditions:

- You have a license for Parallel Computing Toolbox™ software.

- Parallel computing is enabled with matlabpool, a Parallel Computing Toolbox function.

- The following options are set using psoptimset:

  - Cache is 'off' (default)

  - CompletePoll is 'on'

  - Vectorized is 'off' (default)

  - UseParallel is 'always'

When these conditions hold, the solver computes the objective function and constraint values of the pattern search in parallel during a poll.

Currently, UseParallel cannot be accessed using the Optimization Tool GUI.

Using Parallel Computing with patternsearch

- "Using Parallel Computing with Multicore Processors" on page 5-49
- "Using Parallel Computing with a Multiprocessor Network" on page 5-49

Using Parallel Computing with Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a matlabpool of up to 4 parallel workers with a basic Parallel Computing Toolbox license.

Suppose you have a dual-core processor, and wish to use parallel computing:

- Enter

  ```
  matlabpool open 2
  ```

  at the command line. The 2 specifies the number of processors to use.
- Enter

  ```
  options = psoptimset('UseParallel','always');
  ```

When you run an applicable solver with options, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set UseParallel to 'never'. To halt all parallel computation, enter

```
matlabpool close
```

Using Parallel Computing with a Multiprocessor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB® Distributed Computing Server™ software to establish parallel computation. Here are the steps to take:

1 Obtain a license for Parallel Computing Toolbox functions and MATLAB Distributed Computing Server software.

2 Configure your system for parallel processing. See the Parallel Computing Toolbox documentation, or *MATLAB Distributed Computing Server System Administrator's Guide*.

In particular, if *network_file_path* is the network path to your objective or constraint functions, enter

```
pctRunOnAll('addpath network_file_path')
```

so the worker processors can access your objective or constraint M-files.

Check whether an M-file is on the path of every worker by entering

```
pctRunOnAll('which filename')
```

If any worker does not have a path to the M-file, it reports

```
filename not found.
```

3 At the command line enter

```
matlabpool open conf
```

or

```
matlabpool open conf n
```

where `conf` is your configuration, and `n` is the number of processors you wish to use.

4 Enter

```
options = psoptimset('UseParallel','always');
```

Once your parallel computing environment is established, applicable solvers automatically use parallel computing whenever called with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `'never'`. To halt all parallel computation, enter

```
matlabpool close
```

Parallel Search Function

`patternsearch` can optionally call a search function at each iteration. The search is done in parallel under the following conditions:

- `CompleteSearch` is `'on'`.
- The search method is not `@searchneldermead` or `custom`.
- If the search method is a pattern search function or Latin hypercube search, `UseParallel` is `'always'`.
- If the search method is `ga`, the search method option structure has `UseParallel` set to `'always'`.

Implementation Issues in Parallel Pattern Search

Pattern search is implemented in the `patternsearch` solver by using the Parallel Computing Toolbox function `parfor`. `parfor` distributes the evaluation of objective and constraint functions among multiple processes or processors.

The limitations on options, listed in "Parallel Pattern Search" on page 5-48, arise partly from the limitations of `parfor`, and partly from the nature of parallel processing:

- `Cache` is implemented as a persistent variable internally. `parfor` does not handle persistent variables, because the variable could be set differently at different processors.
- `CompletePoll` determines whether a poll stops as soon as a better point is found. When searching in parallel with `parfor`, all evaluations are scheduled at once, and computing continues after all evaluations are returned. It is not easy to halt evaluations once they have been scheduled.
- `Vectorized` determines whether a pattern is evaluated with one function call. If it is `'on'`, it is not possible to distribute the evaluation of the function using `parfor`.

More caveats related to `parfor` are listed in the "Limitations" section of the Parallel Computing Toolbox documentation.

Parallel Computing Considerations

The "Improving Performance with Parallel Computing" section of the Optimization Toolbox™ documentation contains information on factors that affect the speed of parallel computations, factors that affect the results of parallel computations, and searching for global optima. Those considerations also apply to parallel computing with pattern search.

Additionally, there are considerations having to do with random numbers that apply to Genetic Algorithm and Direct Search Toolbox functions. Random number sequences in MATLAB® are pseudorandom, determined from a "seed," an initial setting. Parallel computations use seeds that are not necessarily controllable or reproducible. For example, there is a default global setting on each instance of MATLAB that determines the current seed for random sequences.

Parallel pattern search does not have reproducible polls when used with MADS, and does not have reproducible searches with MADS, the genetic algorithm, or Latin hypercube search methods.

6

Using the Genetic Algorithm

Genetic Algorithm Optimizations Using the Optimization Tool GUI

Introduction

The Optimization Tool GUI is described in the chapter Optimization Tool in the *Optimization Toolbox™ User's Guide*. This section describes some places where there are some differences between the use of the genetic algorithm in the Optimization Tool and the use of other solvers.

Displaying Plots

The **Plot functions** pane, shown in the following figure, enables you to display various plots of the results of the genetic algorithm.
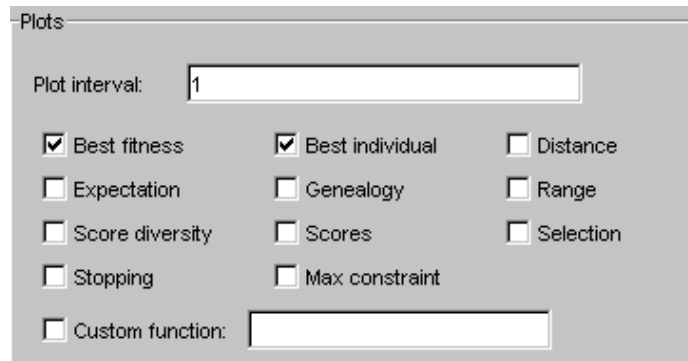
Select the check boxes next to the plots you want to display. For example, if you select **Best fitness** and **Best individual**, and run the example described in "Example: Rastrigin's Function" on page 3-8, the tool displays plots similar to those shown in the following figure.
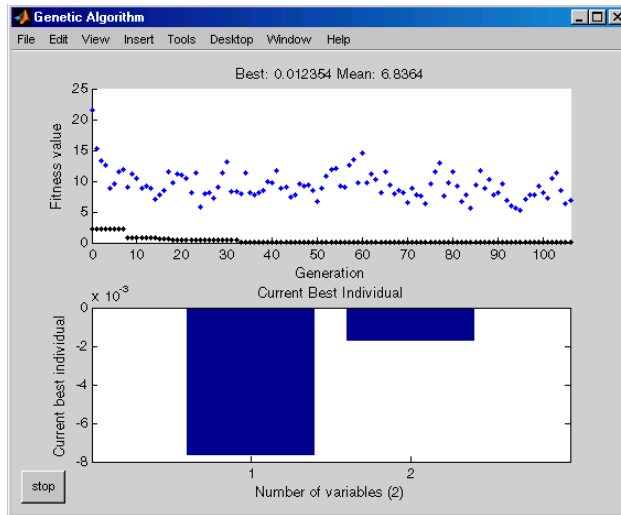
The upper plot displays the best and mean fitness values in each generation. The lower plot displays the coordinates of the point with the best fitness value in the current generation.

Note When you display more than one plot, clicking on any plot while the genetic algorithm is running or after the solver has completed opens a larger version of the plot in a separate window.

"Plot Options" on page 9-24 describes the types of plots you can create.

Example — Creating a Custom Plot Function

If none of the plot functions that come with the software 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 Custom 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 logarithmic 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 **Plot functions** 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 3-8, the tool displays plots similar to those shown in the following figure.
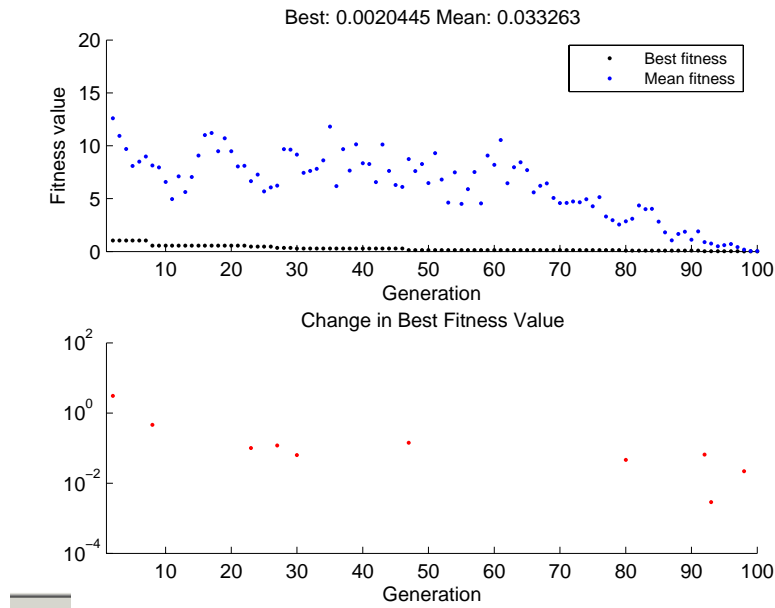
Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot only shows changes that are greater then 0. The logarithmic scale enables you to see small changes in the fitness function that the upper plot does not reveal.

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.Generations` 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 9-26.

- `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 Optimization 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. After the algorithm has run, you can clear your results using the **Clear Status** button in the **Run solver** settings.

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.

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 Optimization Tool:

- Set **Fitness function** to @ackleyfcn, which computes Ackley's function, a test function provided with the software.
- Set **Number of variables** to 10.
- Select **Best fitness** in the **Plot functions** pane.
- Click **Start**.

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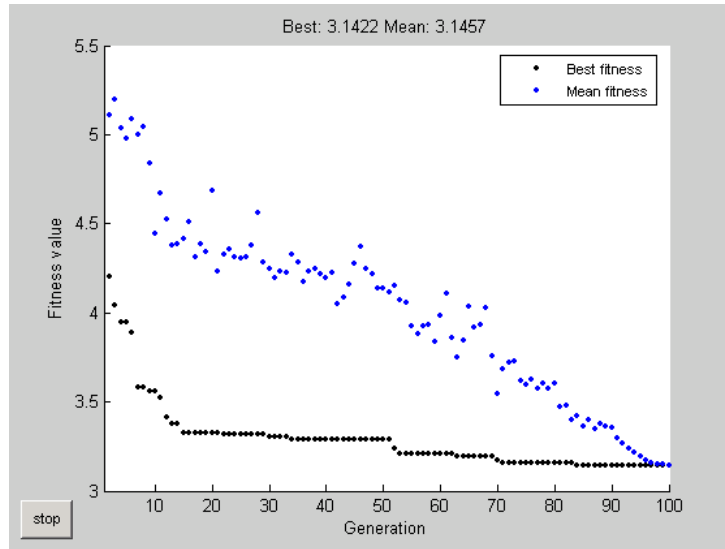
Suppose you want to experiment by running the genetic algorithm with other options settings, and then later restart this run from its final population with its current options settings. You can do this using the following steps:

1 Click **Export to Workspace**.

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_uniform, in the text field.

- Select **Include information needed to resume this run**.

The dialog box should now appear as in the following figure.
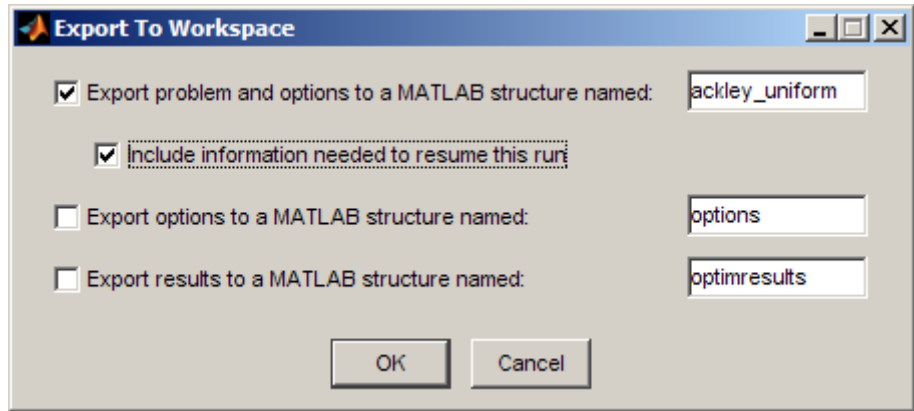
3 Click **OK**.

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
         nvars: 10
         Aineq: []
         bineq: []
           Aeq: []
           beq: []
            lb: []
            ub: []
       nonlcon: []
     randstate: []
    randnstate: []
        solver: 'ga'
       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.
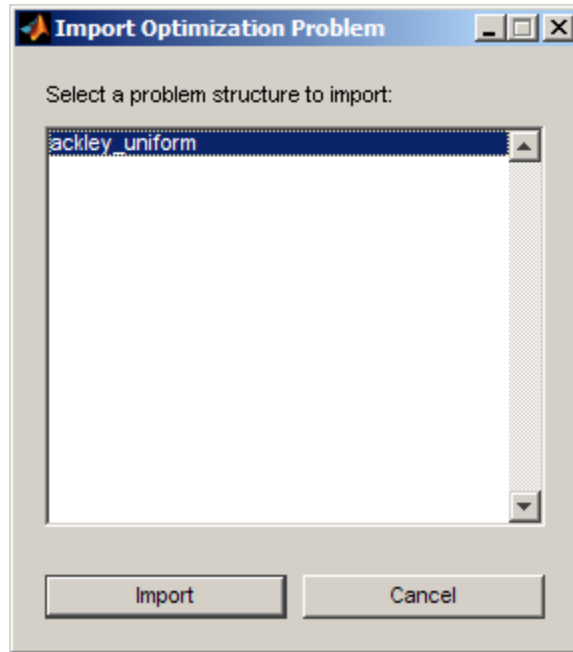
2 Select ackley_uniform.

3 Click **Import**.

This sets the **Initial population** and **Initial scores** fields in the **Population** panel 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.
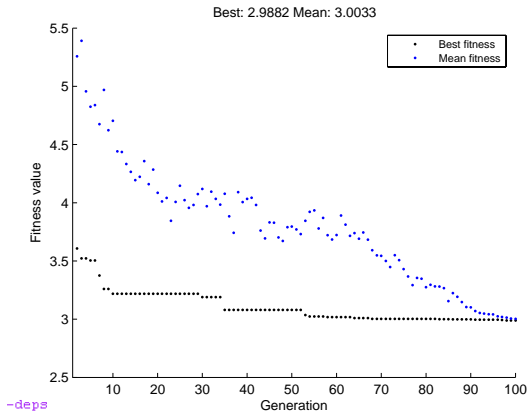
First run
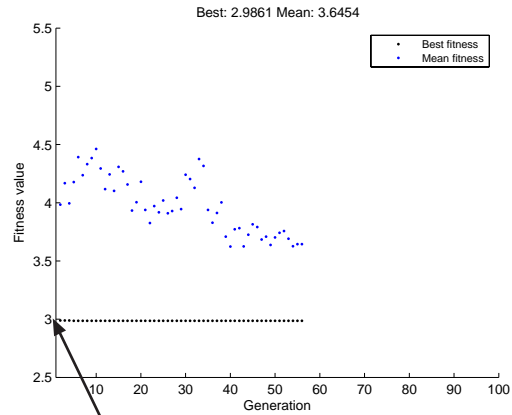
Run resumes here

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.

Using the Genetic Algorithm from the Command Line

In this section...

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 input and output arguments, see the reference page for ga.

You can run the example described in "Example: Rastrigin's Function" on page 3-8 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 exitflag output population scores] = ga(@fitnessfcn, nvars)
```

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

- exitflag — Integer value corresponding to the 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 ga reference page for more information about these arguments.

Setting Options for ga at the Command Line

You can specify any of the options that are available for ga by passing an options structure as an input argument to ga using the syntax

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

This syntax does not specify any linear equality, linear inequality, or nonlinear constraints.

You create the options structure using the function gaoptimset.

```
options = gaoptimset(@ga)
```

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
          ParetoFraction: []
       MigrationDirection: 'forward'
        MigrationInterval: 20
        MigrationFraction: 0.2000
              Generations: 100
                TimeLimit: Inf
              FitnessLimit: -Inf
            StallGenLimit: 50
           StallTimeLimit: Inf
                   TolFun: 1.0000e-006
                   TolCon: 1.0000e-006
        InitialPopulation: []
            InitialScores: []
           InitialPenalty: 10
            PenaltyFactor: 100
              PlotInterval: 1
               CreationFcn: @gacreationuniform
        FitnessScalingFcn: @fitscalingrank
             SelectionFcn: @selectionstochunif
             CrossoverFcn: @crossoverscattered
               MutationFcn: {[1x1 function_handle]  [1]  [1]}
        DistanceMeasureFcn: []
                 HybridFcn: []
                   Display: 'final'
                  PlotFcns: []
                OutputFcns: []
                Vectorized: 'off'
                UseParallel: 'never'
```

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 Optimization Tool

As an alternative to creating an options structure using `gaoptimset`, you can set the values of options in the Optimization Tool and then export the options to a structure in the MATLAB® workspace, as described in the "Importing and Exporting Your Work" section of the Optimization Toolbox™ documentation. If you export the default options in the Optimization Tool, the resulting structure `options` has the same settings as the default structure returned by the command

```
options = gaoptimset(@ga)
```

except that the option `'Display'` defaults to `'off'` in an exported structure, and is `'final'` in the default at the command line.

If you export a problem structure, `ga_problem`, from the Optimization 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
- `Aineq` — Matrix for inequality constraints
- `Bineq` — Vector for inequality constraints
- `Aeq` — Matrix for equality constraints
- `Beq` — Vector for equality constraints
- `LB` — Lower bound on `x`
- `UB` — Upper bound on `x`
- `nonlcon` — Nonlinear constraint function
- `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 exitflag 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('twister', output.randstate);
randn('state', output.randnstate);
```

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

You can reproduce your run in the Optimization Tool by checking the box **Use random states from previous run** in the **Run solver and view results** section.
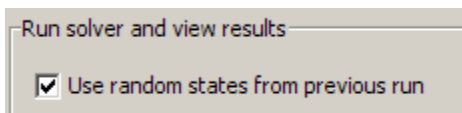
Note If you do not need to reproduce your results, it is better not to set the states of rand and randn, so that you get the benefit of the randomness in the genetic algorithm.

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,exitflag,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,exitflag,output,final_pop2] = ...
  ga(@fitnessfcn,nvars,[],[],[],[],[],[],[],options);
```

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

In Optimization Tool, you can choose to export a problem in a way that lets you resume the run. Simply check the box **Include information needed to resume this run** when exporting the problem.
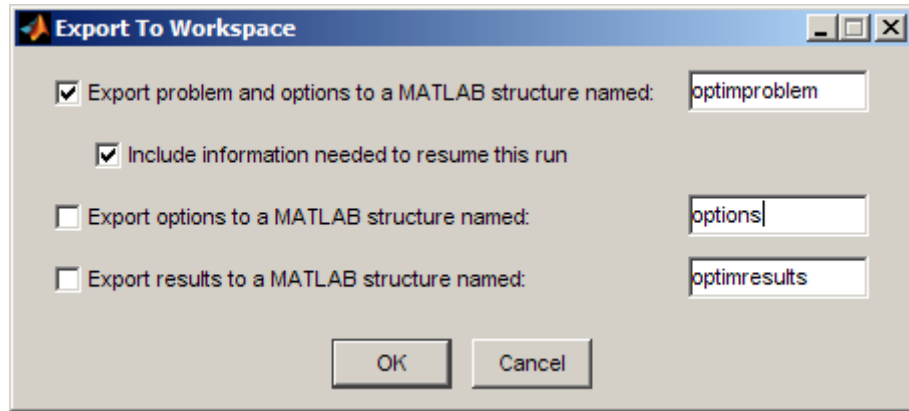
This saves the final population, which becomes the initial population when imported.

If you want to run a problem that was saved with the final population, but would rather not use the initial population, simply delete or otherwise change the initial population in the **Options > Population** pane.

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 21 times, varying options.CrossoverFraction from 0 to 1 in increments of 0.05, and records the results.

```
options = gaoptimset('Generations',300);
rand('twister', 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')
```

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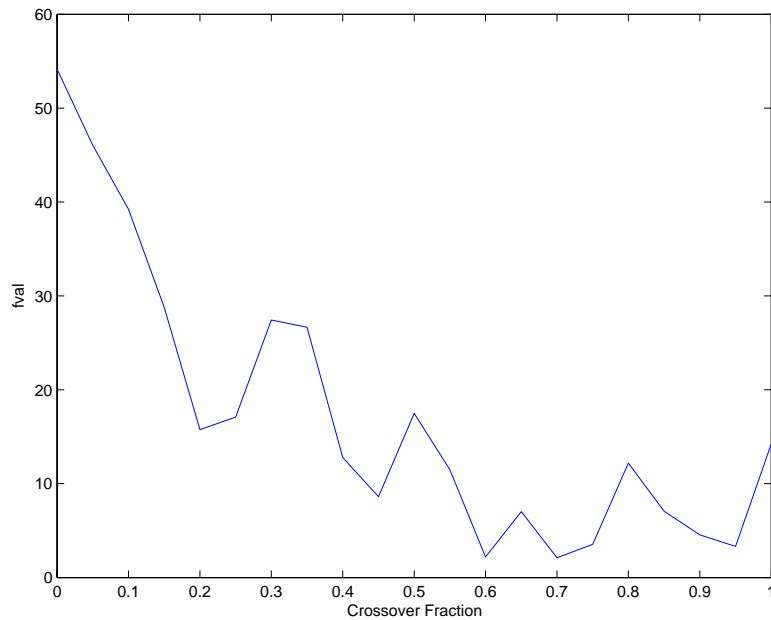
The plot indicates that you get the best results by setting `options.CrossoverFraction` to a value somewhere between `0.6` and `0.95`.

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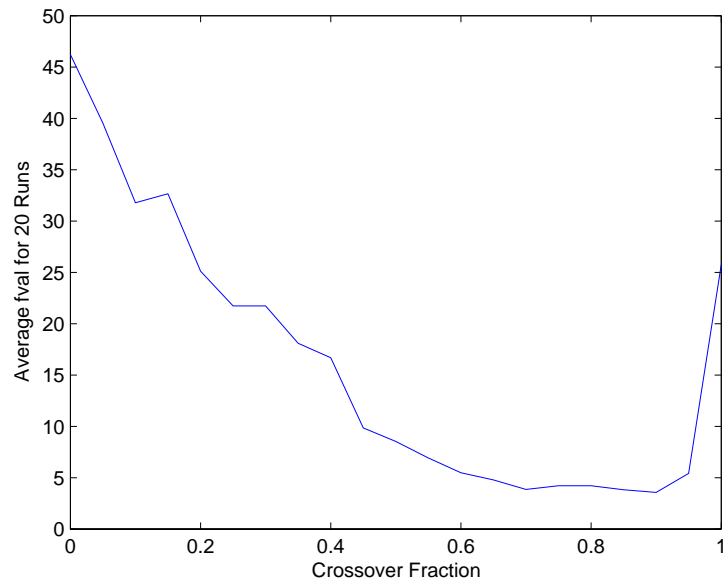
The plot narrows the range of best choices for options.CrossoverFraction to values between 0.7 and 0.9.

Genetic Algorithm Examples

In this section...

Improving Your Results

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 9-23.

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 6-36 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 Optimization Tool creates a random initial population using a 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 by specifying the lower and upper bounds. Subsequent generations can contain points whose entries do not lie in the initial range. Additionally, the upper and lower bounds can be adjusted by setting the **Bounds** fields in the **Constraints** panel.

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 3-8. The minimum value of the function is 0, which occurs at the origin.

To run the example, make the following settings in the Optimization Tool:

- Set **Fitness function** to `@rastriginsfcn`.
- Set **Number of variables** to 2.
- Select **Best fitness** in the **Plot functions** pane.
- Select **Distance** in the **Plot functions** 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.
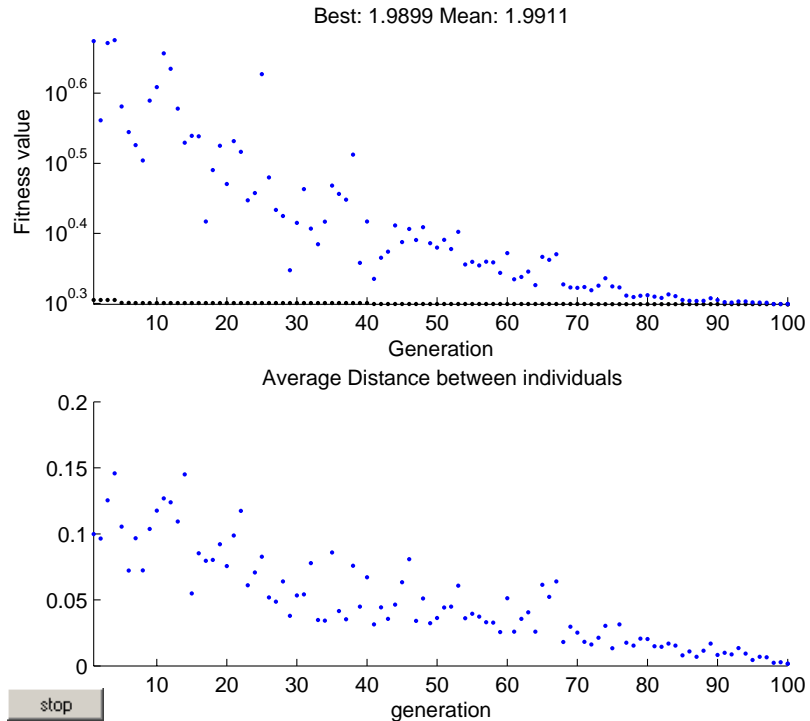
The upper plot, which displays the best fitness at each generation, shows little progress in lowering the fitness value. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too little diversity for the algorithm to make progress.

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

Best: 3.8861 Mean: 10.2159
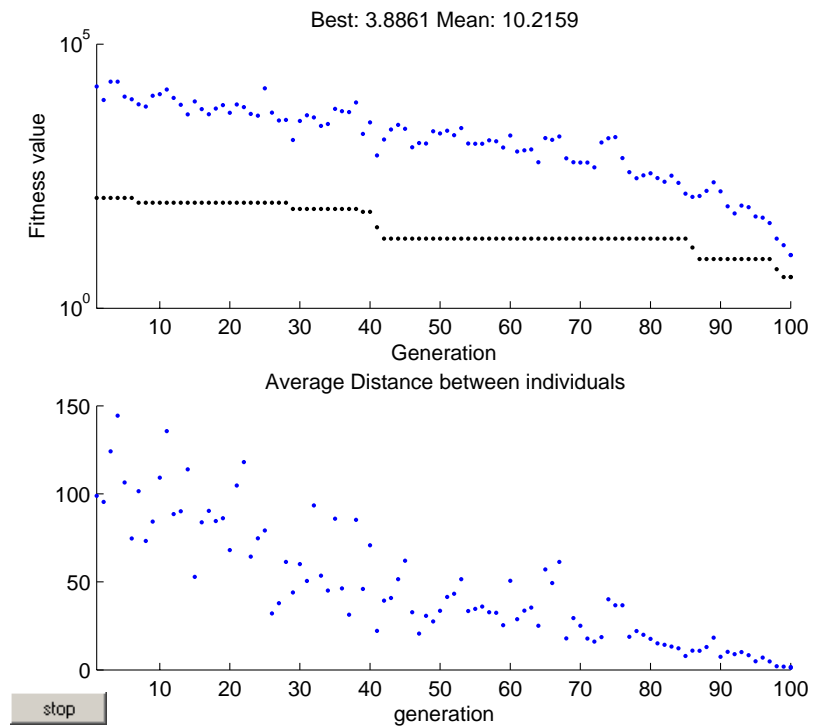
This time, the genetic algorithm makes progress, but because the average distance between individuals is so large, the best individuals are far from the optimal solution.

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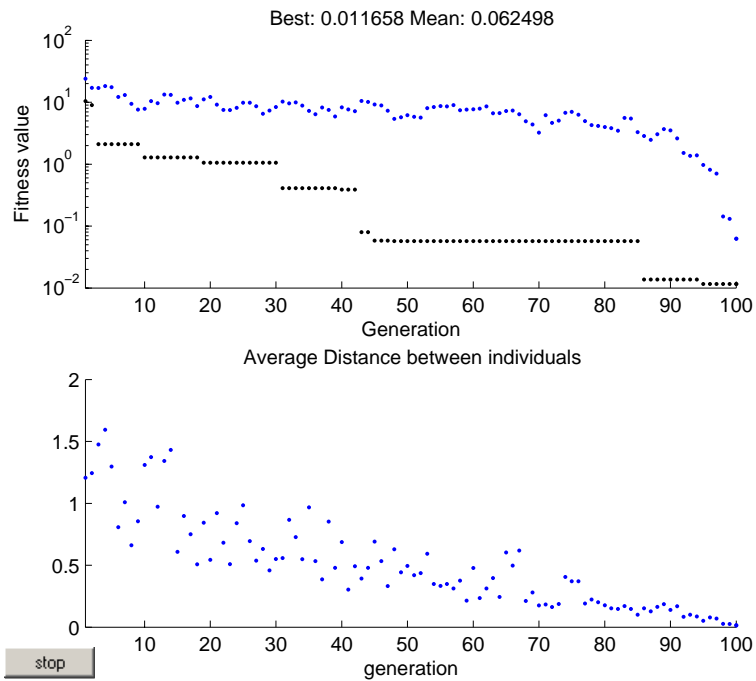
The diversity in this case is better suited to the problem, so the genetic algorithm returns a much better result than in the previous two cases.

Example — Linearly Constrained Population and Custom Plot Function

This example shows how the default creation function for linearly constrained problems, gacreationlinearfeasible, creates a well-dispersed population that satisfies linear constraints and bounds. It also contains an example of a custom plot function.

The problem uses the objective function in lincontest6.m, a quadratic:

$$f(x) = \frac{x_1^2}{2} + x_2^2 - x_1 x_2 - 2x_1 - 6x_2.$$

To see code for the function, enter `type lincontest6`. The constraints are three linear inequalities:

$$x_1 + x_2 \leq 2,$$
$$-x_1 + x_2 \leq 2,$$
$$2x_1 + 2x_2 \leq 3.$$

Also, the variables x_i are restricted to be positive.

1 Create a custom plot function file containing the following code:

```
function state = gaplotshowpopulation2(unused,state,flag,fcn)
% This plot function works in 2-d only
if size(state.Population,2) > 2
    return;
end
if nargin < 4
    fcn = [];
end
% Dimensions to plot
dimensionsToPlot = [1 2];

switch flag
    % Plot initialization
    case 'init'
        pop = state.Population(:,dimensionsToPlot);
        plotHandle = plot(pop(:,1),pop(:,2),'*');
        set(plotHandle,'Tag','gaplotshowpopulation2')
        title('Population plot in two dimension','interp','none')
        xlabelStr = sprintf('%s %s','Variable ',...
                    num2str(dimensionsToPlot(1)));
        ylabelStr = sprintf('%s %s','Variable ',...
                    num2str(dimensionsToPlot(2)));
        xlabel(xlabelStr,'interp','none');
        ylabel(ylabelStr,'interp','none');
        hold on;

        % plot the inequalities
        plot([0 1.5],[2 0.5],'m-.') %  x1 + x2 <= 2
        plot([0 1.5],[1 3.5/2],'m-.'); % -x1 + 2*x2 <= 2
```

```
            plot([0 1.5],[3 0],'m-.'); % 2*x1 + x2 <= 3
            % plot lower bounds
            plot([0 0], [0 2],'m-.'); % lb = [ 0 0];
            plot([0 1.5], [0 0],'m-.'); % lb = [ 0 0];
            set(gca,'xlim',[-0.7,2.2])
            set(gca,'ylim',[-0.7,2.7])

            % Contour plot the objective function
            if ~isempty(fcn)
                range = [-0.5,2;-0.5,2];
                pts = 100;
                span = diff(range')/(pts - 1);
                x = range(1,1): span(1) : range(1,2);
                y = range(2,1): span(2) : range(2,2);

                pop = zeros(pts * pts,2);
                values = zeros(pts,1);
                k = 1;
                for i = 1:pts
                    for j = 1:pts
                        pop(k,:) = [x(i),y(j)];
                        values(k) = fcn(pop(k,:));
                        k = k + 1;
                    end
                end
                values = reshape(values,pts,pts);
                contour(x,y,values);
                colorbar
            end
            % Pause for three seconds to view the initial plot
            pause(3);
        case 'iter'
            pop = state.Population(:,dimensionsToPlot);
            plotHandle = findobj(get(gca,'Children'),'Tag',...
                        'gaplotshowpopulation2');
            set(plotHandle,'Xdata',pop(:,1),'Ydata',pop(:,2));
    end
```

The custom plot function plots the lines representing the linear inequalities and bound constraints, plots level curves of the objective (fitness) function, and plots the population as it evolves.

2 At the command line, enter the constraints as a matrix and vectors:

```
A = [1,1;-1,2;2,1]; b = [2;2;3]; lb = zeros(2,1);
```

3 Set options to use gaplotshowpopulation:

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

4 Run the optimization using options:

```
[x,fval] = ga(@lincontest6,2,A,b,[],[],lb,[],[],options);
```

A plot window appears showing the linear constraints, bounds, level curves of the objective function, and initial distribution of the population:
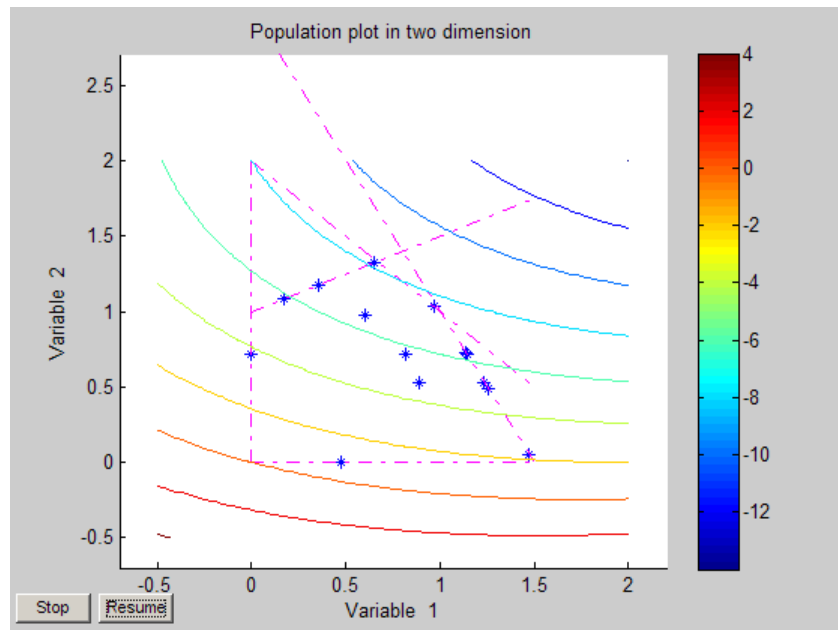
You can see that the initial population is biased to lie on the constraints.

The population eventually concentrates around the minimum point:
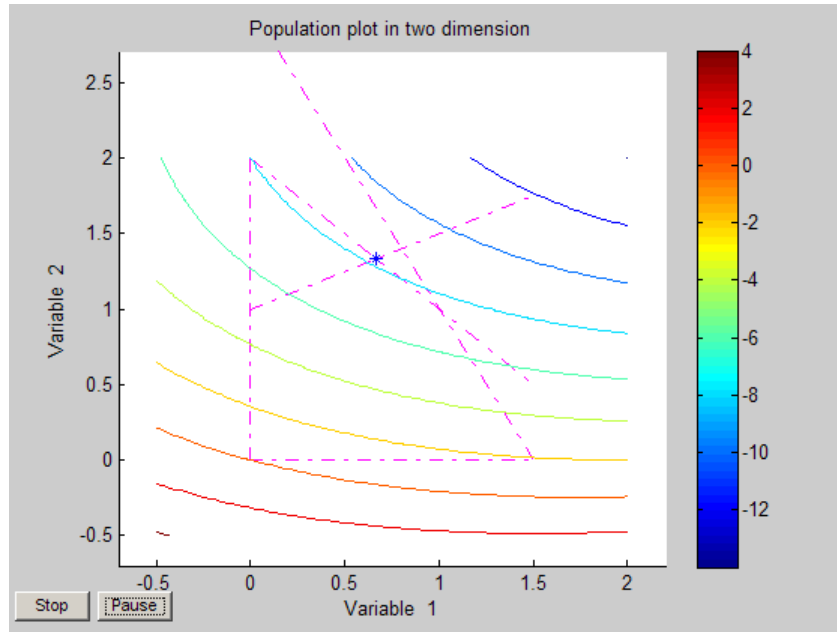
Setting the Population Size

The **Population 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 **Population 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 option, 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
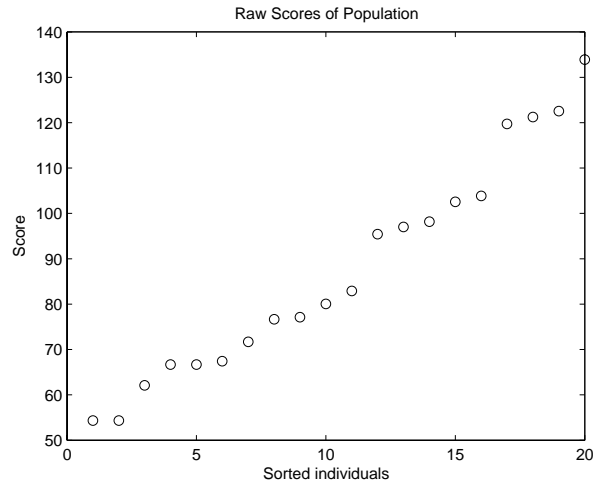
The following plot shows the scaled values of the raw scores using rank scaling.

Scaled Values Using Rank Scaling
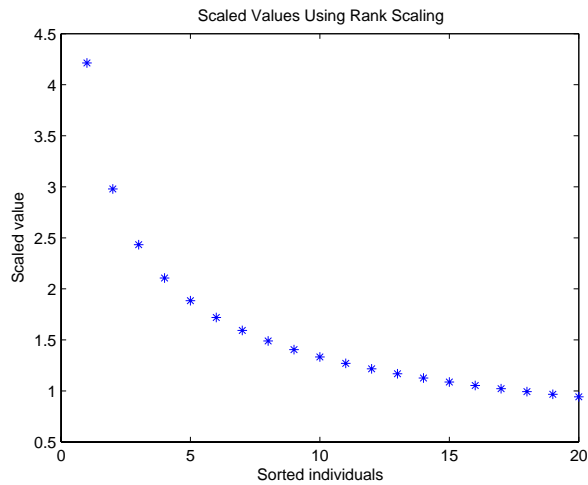
Because the algorithm minimizes the fitness function, lower raw scores have higher scaled values. Also, because rank scaling assigns values that depend

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 options, such as Top. By default, top scaling assigns 40 percent of the fittest individuals to the same scaled value and assigns the rest of the individuals to value 0. Using the default selection function, only 40 percent of the 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.

Comparison of Rank and Top Scaling
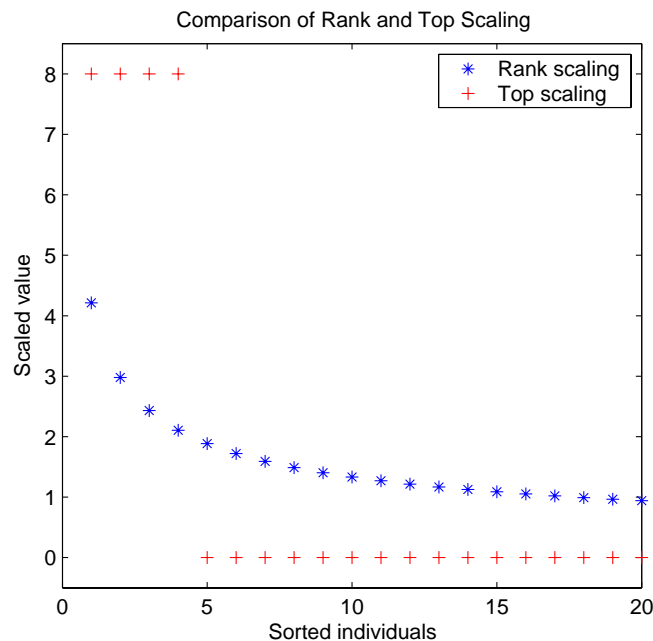
Because top scaling restricts parents to the fittest individuals, it creates less diverse populations than rank scaling. The following plot compares the

variances of distances between individuals at each generation using rank and top scaling.

Variance of Distance Between Individuals Using Rank and Top Scaling
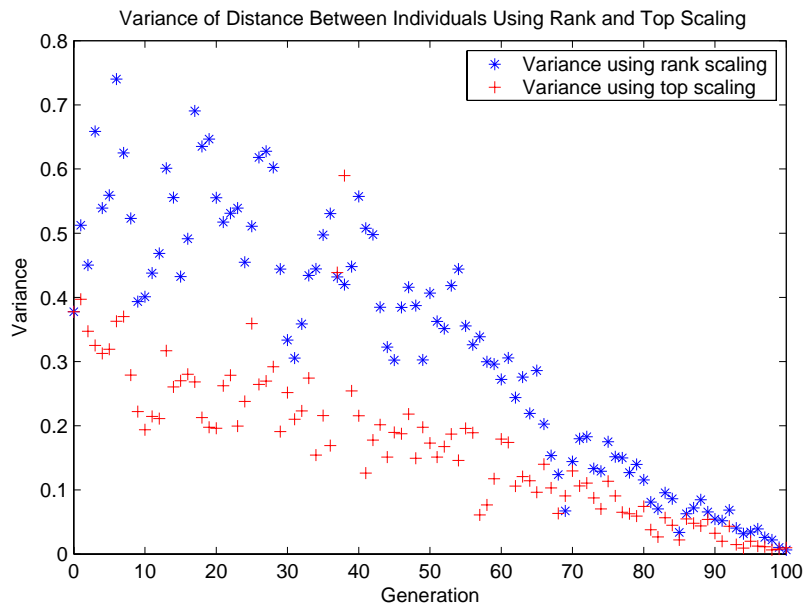
Selection

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 option, `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 option 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 6-38 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.

See "Creating the Next Generation" on page 3-22 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 are as follows:

- There are two 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 four individuals, other than elite children, are mutation children.

Setting the Amount of Mutation

The genetic algorithm applies mutations using the option that you specify on the **Mutation function** pane. The default mutation option, 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 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 options **Distance** and **Range**, and then running the genetic algorithm on a problem such as the one described in "Example: Rastrigin's Function" on page 3-8. The following figure shows the plot.
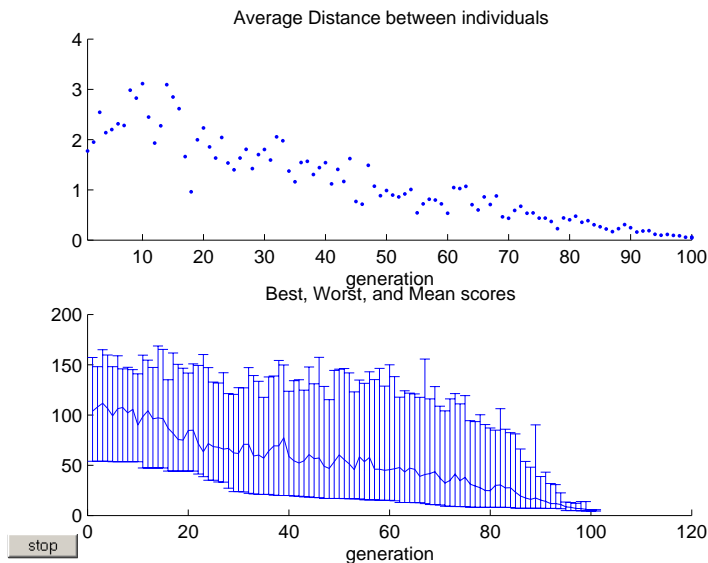
The upper plot displays the average distance between points in each generation. As the amount of mutation decreases, so does the average distance between individuals, which is approximately 0 at the final generation. The lower plot displays a vertical line at each generation, showing the range from the 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.
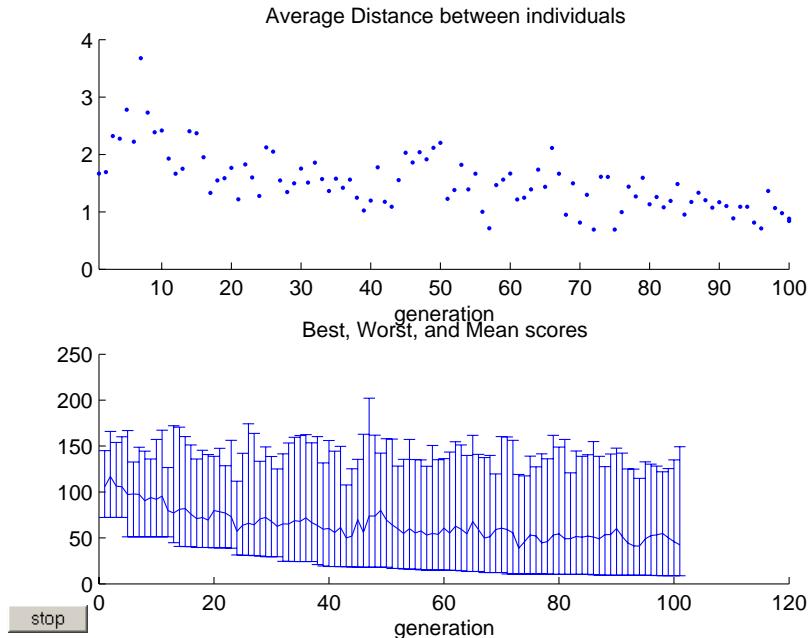
With **Shrink** set to 0.5, the average amount of mutation decreases by a factor of 1/2 by the final generation. As a result, the average distance between individuals decreases by approximately the same factor.

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,

$$f(x_1, x_2, ..., x_n) = |x_1| + |x_2| + ... + |x_n|$$

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 **Best fitness** and **Distance** in the **Plot functions** pane.

Run the example with the default value of `0.8` for **Crossover fraction**, in the **Options > Reproduction** pane. This returns the best fitness value of approximately `0.2` and displays the following plots.
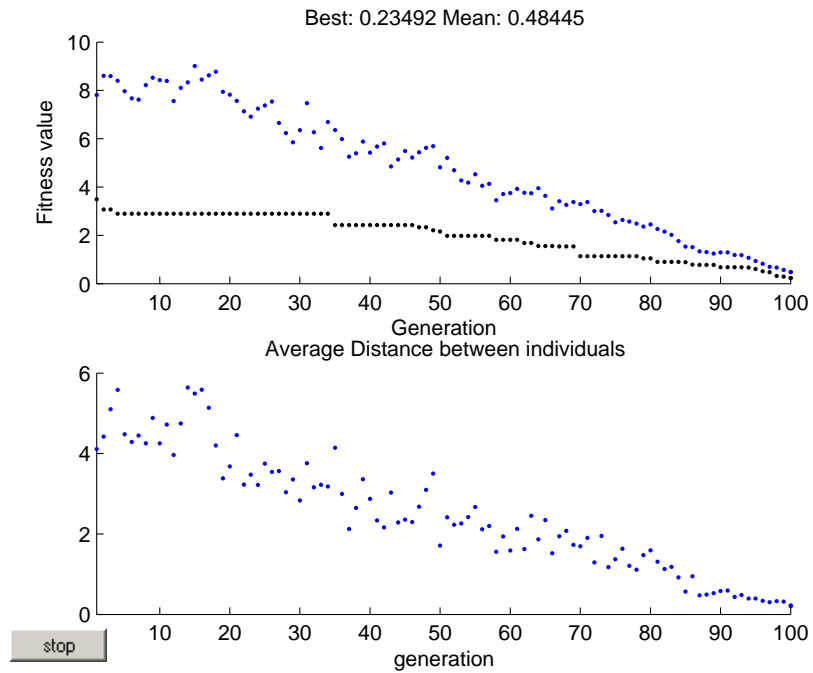
Crossover Without Mutation

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.

Best: 1.3161 Mean: 1.3161
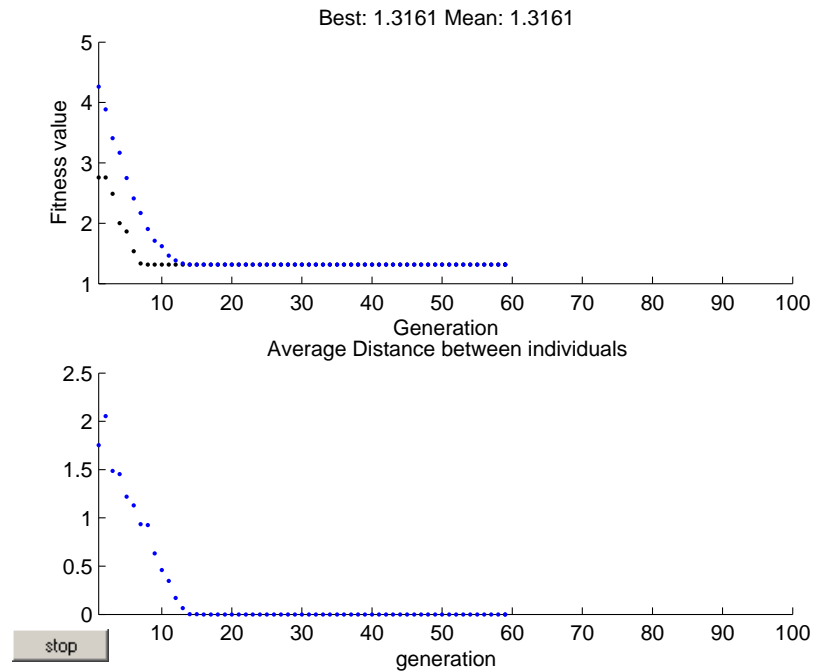
In this case, the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation. The algorithm generates the best individual that it can using these genes at generation number 8, where the best fitness 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.
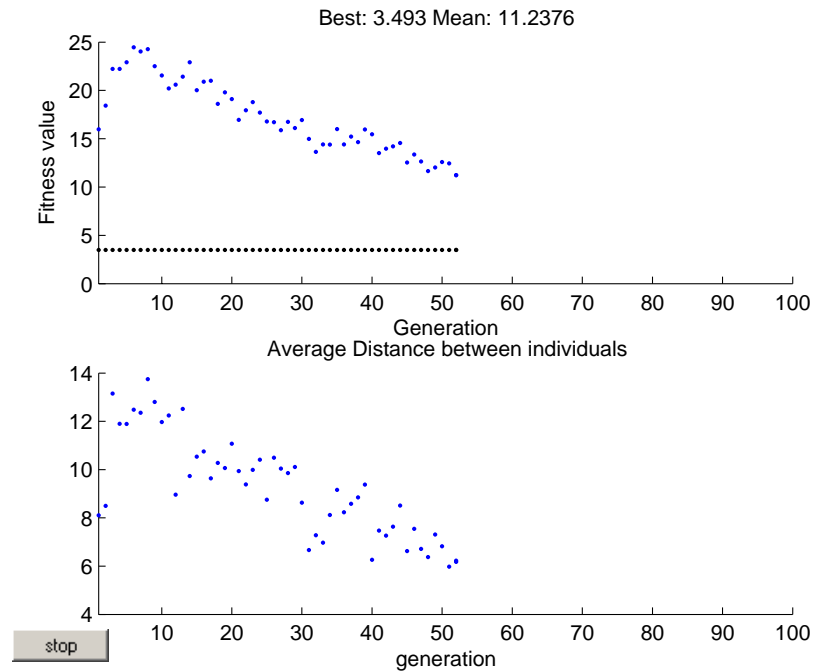
In this case, the random changes that the algorithm applies never improve the fitness value of the best individual at the first generation. While it improves the individual genes of other individuals, as you can see in the upper plot by the decrease in the mean value of the fitness function, these improved 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 software, 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.
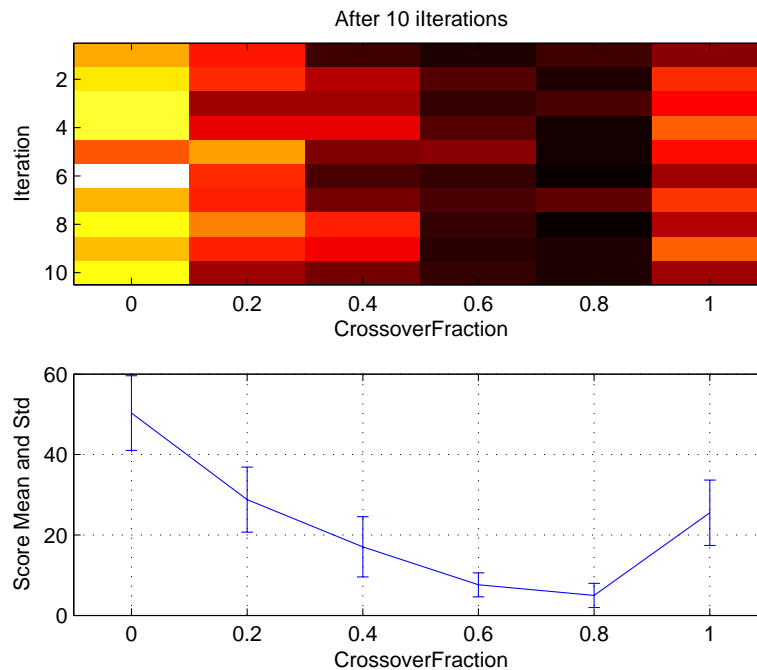
The lower plot shows the means and standard deviations of the best fitness values over 10 generations, for each of the values of the crossover fraction. The upper plot shows a color-coded display of the best fitness values in each generation.

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.

Global vs. 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

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

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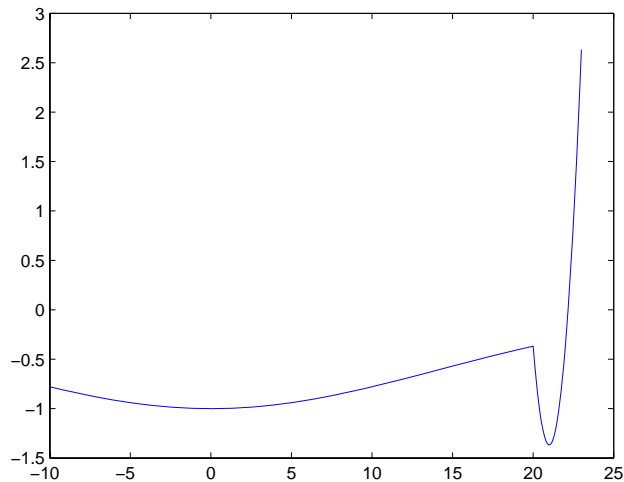
The function has two local minima, one at $x = 0$, where the function value is -1, and the other at $x = 21$, where the function value is $-1 - 1/e$. Since the latter value is smaller, the global minimum occurs at $x = 21$.

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 Optimization 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.
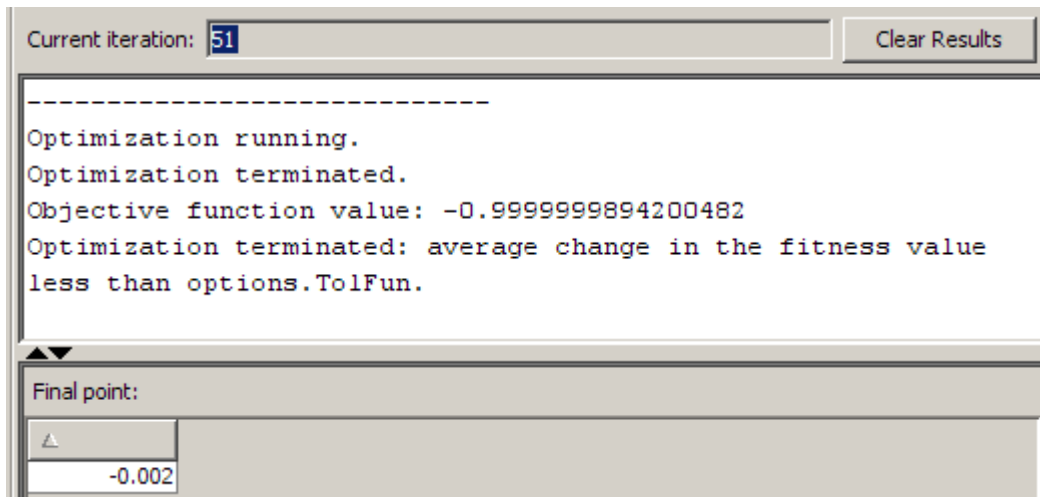
The following custom plot shows why the algorithm finds the local minimum rather than the global minimum. The plot shows the range of individuals in each generation and the best individual.
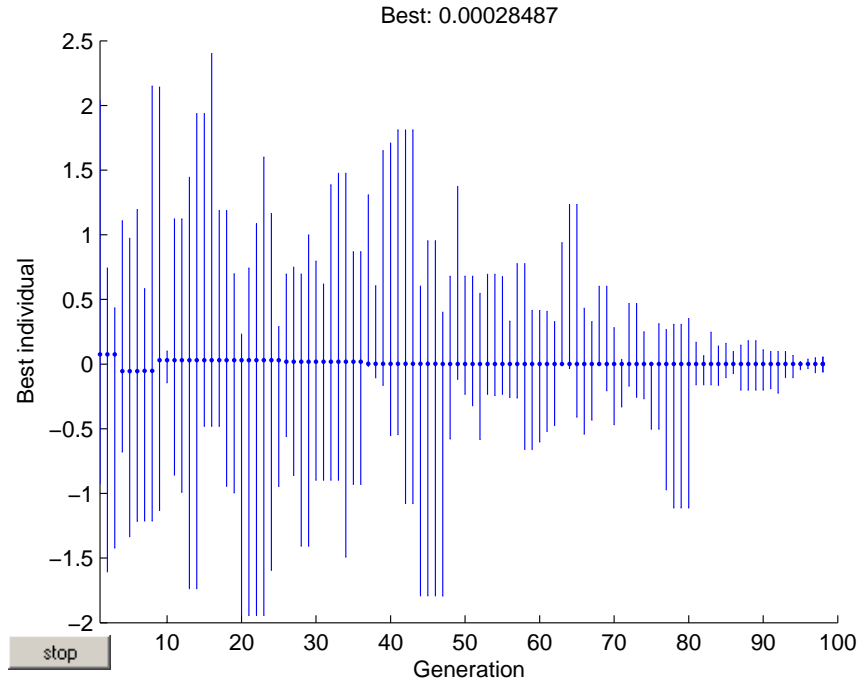
Note that all individuals are between -2 and 2.5. While this range is larger than the default **Initial range** of [0;1], due to mutation, it is not large enough to explore points near the global minimum at $x = 21$.

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.

Then click **Start**. The genetic algorithm returns a point very close to 21.
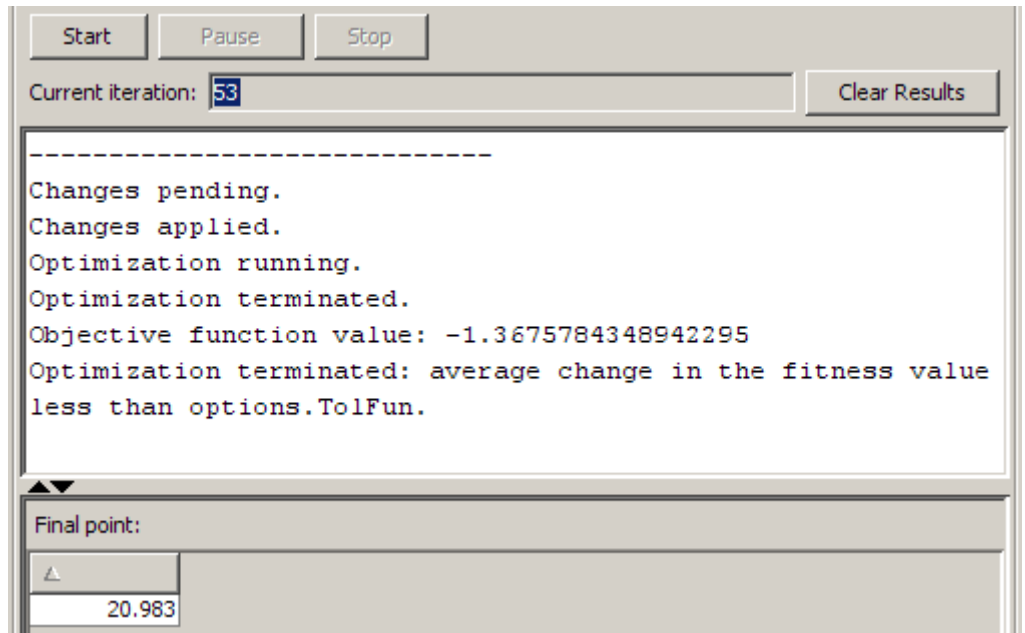
This time, the custom plot shows a much wider range of individuals. By the second generation there are individuals greater than 21, and by generation 12, the algorithm finds a best individual that is approximately equal to 21.
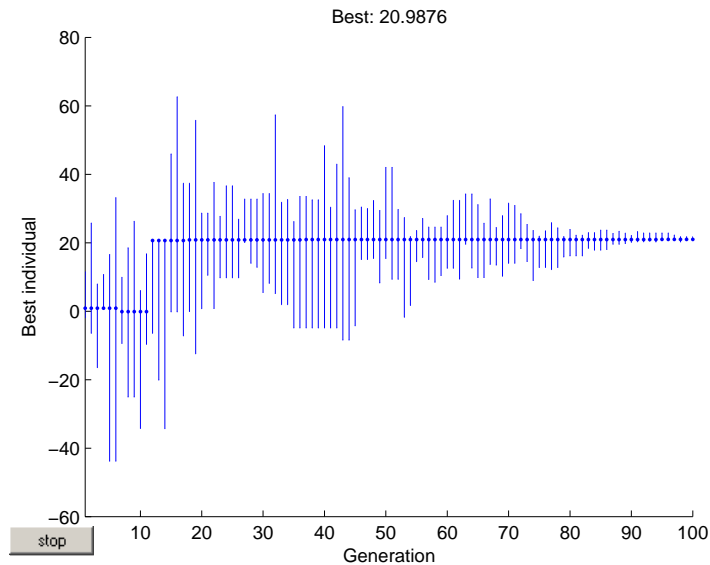
Using a Hybrid Function

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 Optimization Toolbox™ function `fminunc`, an unconstrained minimization function. 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

$$f(x_1, x_2) = 100 \left(x_2 - x_1^2 \right)^2 + (1 - x_1)^2.$$

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

Minimum at (1,1)
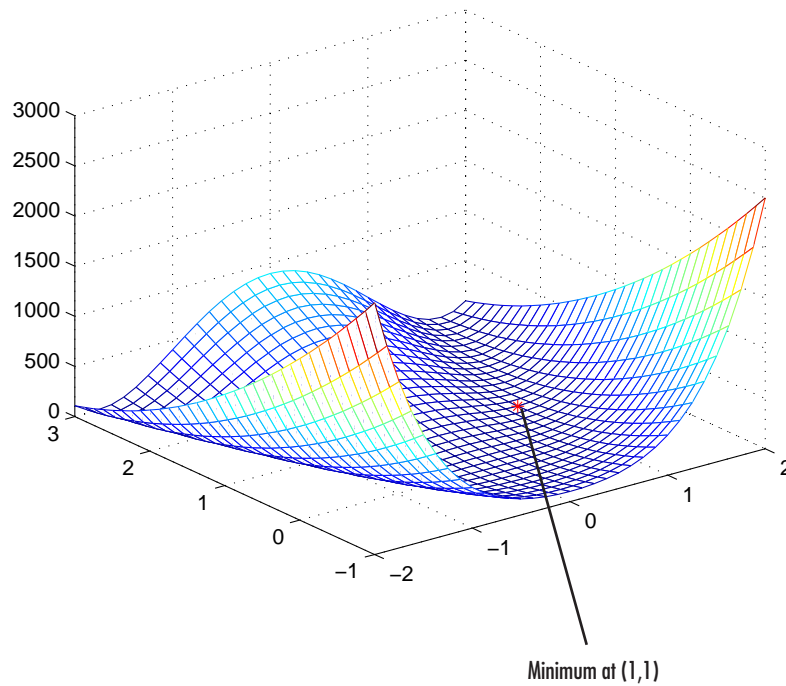
The software provides an M-file, dejong2fcn.m, that computes Rosenbrock's function. To see a demo of this example, enter

```
hybriddemo
```

at the MATLAB prompt.

To explore the example, first enter optimtool('ga') to open the Optimization Tool to the ga solver. 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, click **Start** to run the genetic algorithm by itself. The genetic algorithm displays the following results in the **Run solver and view results** pane:
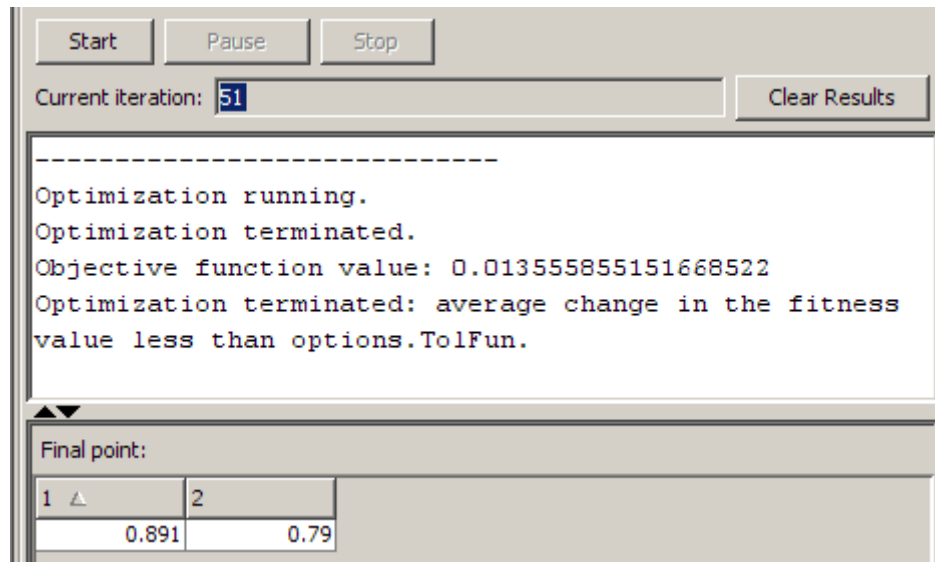
The final point is close to the true minimum at (1, 1). You can improve this result by setting **Hybrid function** to `fminunc` (in the **Hybrid function** options).
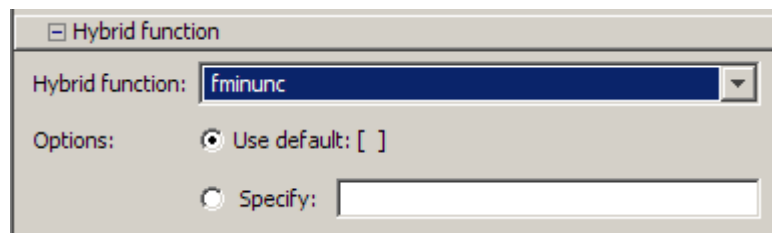
The function `fminunc` uses the final point of the genetic algorithm as its initial point. It returns a more accurate result, as shown in the **Run solver and view results** pane.
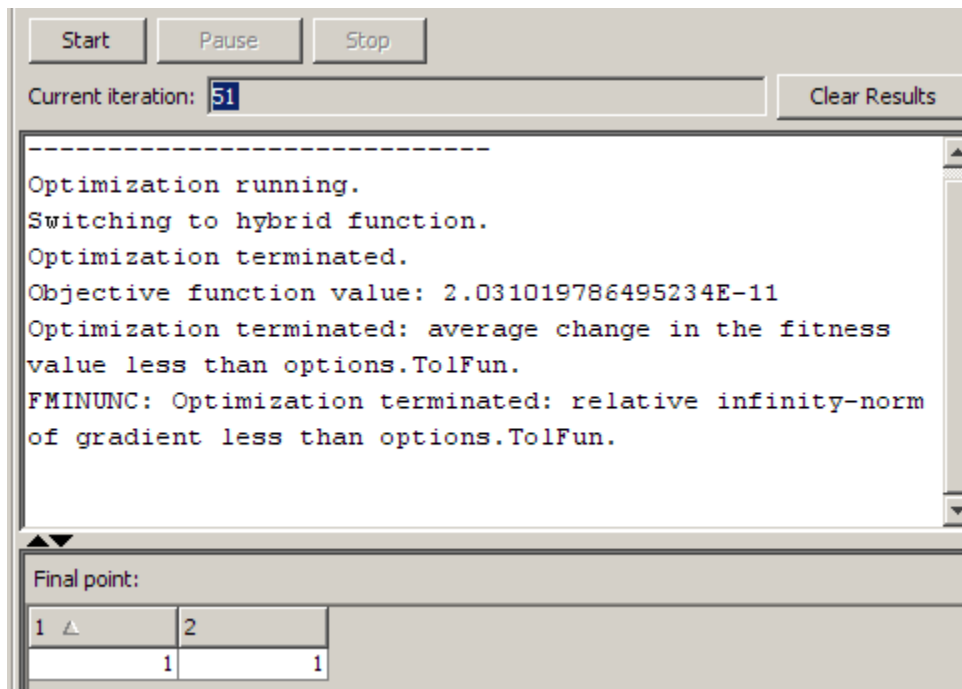
You can set options for the hybrid function separately from the calling function. Use `optimset` or `psoptimset` to create the structure:

```
hybridopts = optimset('display','iter','LargeScale','off');
```

In the Optimization Tool enter the name of your options structure in the **Options** box under **Hybrid function**:
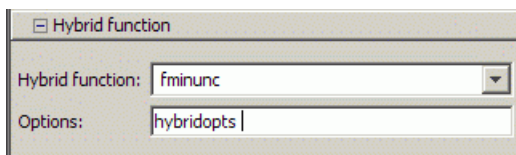
At the command line, the syntax is as follows:

```
options = gaoptimset('HybridFcn',{@fminunc,hybridopts});
```

`hybridopts` must exist before you set `options`.

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 3-24. Increasing the **Generations** option often improves the final result.

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

- Set **Fitness function** to @rastriginsfcn.

- Set **Number of variables** to 10.

- Select **Best fitness** in the **Plot functions** pane.

- Set **Generations** to Inf.

- Set **Stall generations** to Inf.

- Set **Stall time** to Inf.

Run the genetic algorithm for approximately 300 generations and click **Stop**. The following figure shows the resulting best fitness plot after 300 generations.
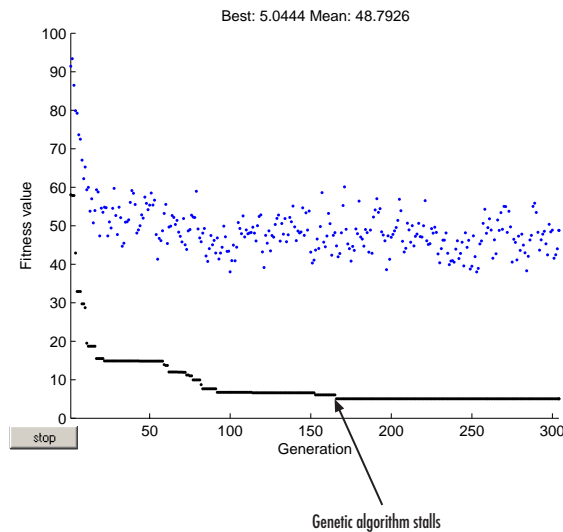
Note that the algorithm *stalls* at approximately generation number 170—that is, there is no immediate improvement in the fitness function after generation 170. If you restore **Stall generations** to its default value of 50, the algorithm would terminate at approximately generation number 230. If the genetic algorithm stalls repeatedly with the current setting for **Generations**, you can try increasing both the **Generations** and **Stall generations** options 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 the value specified in **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.

- In the **Vectorize** pane, set the **Fitness function is vectorized** option to `On`.

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
```

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

Constrained Minimization Using ga

Suppose you want to minimize the simple fitness function of two variables x_1 and x_2,

$$\min_{x} f(x) = 100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$$

subject to the following nonlinear inequality constraints and bounds

$$
\begin{aligned}
&x_1 \cdot x_2 + x_1 - x_2 + 1.5 \leq 0 && \text{(nonlinear constraint)} \\
&10 - x_1 \cdot x_2 \leq 0 && \text{(nonlinear constraint)} \\
&0 \leq x_1 \leq 1 && \text{(bound)} \\
&0 \leq x_2 \leq 13 && \text{(bound)}
\end{aligned}
$$

Begin by creating the fitness and constraint functions. First, create an M-file named `simple_fitness.m` as follows:

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

The genetic algorithm function, `ga`, assumes the fitness function will take one input `x`, where `x` has as many elements as the number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument, `y`.

Then create an M-file, `simple_constraint.m`, containing the constraints

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

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

To minimize the fitness function, you need to pass a function handle to the fitness function as the first argument to the `ga` function, as well as specifying the number of variables as the second argument. Lower and upper bounds are provided as `LB` and `UB` respectively. In addition, you also need to pass a function handle to the nonlinear constraint function.

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

Warning: 'mutationgaussian' mutation function
is for unconstrained minimization only;
using @mutationadaptfeasible mutation function.
Set @mutationadaptfeasible as MutationFcn options
using GAOPTIMSET.
```

```
Optimization terminated: current tolerance on f(x) 1e-007
is less than options.TolFun and constraint violation is
less than options.TolCon.

x =
    0.8122   12.3122

fval =
  1.3578e+004
```

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

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

ga uses the mutation and crossover functions to produce new individuals at every generation. ga satisfies linear and bound constraints by using mutation and crossover functions that only generate feasible points. For example, in the previous call to ga, the default mutation function mutationguassian will not satisfy the linear constraints and so the mutationadaptfeasible is used instead. If you provide a custom mutation function, this custom function must only generate points that are feasible with respect to the linear and bound constraints. All the included crossover functions generate points that satisfy the linear constraints and bounds except the crossoverheuristic function.

Specify mutationadaptfeasible as the mutation function for the minimization problem by using the gaoptimset function.

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

Next run the ga solver.

```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB,ConstraintFunction,options)

Optimization terminated: current tolerance on f(x) 1e-007 is less than options.TolFun
 and constraint violation is less than options.TolCon.

x =
    0.8122   12.3122

fval =
  1.3578e+004
```

Now, use the `gaoptimset` function to create an options structure that will select two plot functions. The first plot function is `gaplotbestf`, which plots the best and mean score of the population at every generation. The second plot function is `gaplotmaxconstr`, which plots the maximum constraint violation of nonlinear constraints at every generation. You can also visualize the progress of the algorithm by displaying information to the command window using the `'Display'` option.

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

Rerun the `ga` solver.

```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],...
             LB,UB,ConstraintFunction,options)
                           Best        max        Stall
Generation  f-count        f(x)     constraint  Generations
    1          849        14915.8         0          0
    2         1567        13578.3         0          0
    3         2334        13578.3         0          1
    4         3043        13578.3         0          2
    5         3752        13578.3         0          3
Optimization terminated: current tolerance on f(x) 1e-009
is less than options.TolFun and constraint violation is
less than options.TolCon.

x =
    0.8122   12.3123

fval =
  1.3578e+004
```
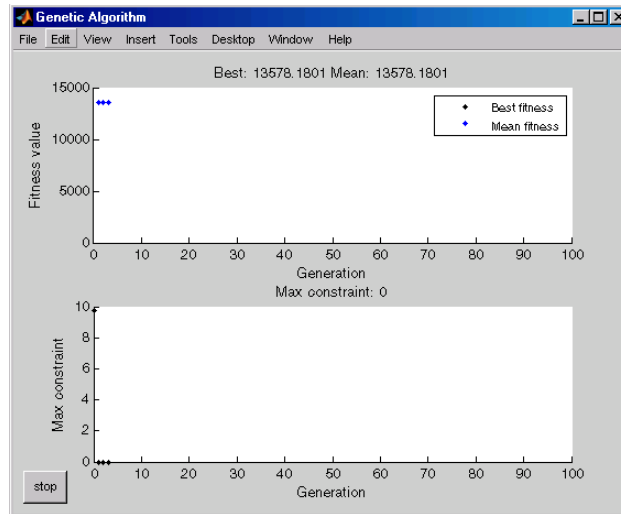
You can provide a start point for the minimization to the ga function by specifying the InitialPopulation option. The ga function will use the first individual from InitialPopulation as a start point for a constrained minimization.

```
X0 = [0.5 0.5]; % Start point (row vector)
options = gaoptimset(options,'InitialPopulation',X0);
```

Now, rerun the ga solver.

```
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],...
             LB,UB,ConstraintFunction,options)
                        Best       max       Stall
Generation  f-count      f(x)    constraint Generations
    1         965       13579.6        0       0
    2        1728       13578.2  1.776e-015    0
    3        2422       13578.2        0       0
Optimization terminated: current tolerance on f(x) 1e-007
is less than options.TolFun and constraint violation is
less than options.TolCon.

x =
    0.8122   12.3122
```

```
fval =
   1.3578e+004
```

Vectorized Constraints

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

"Vectorizing the Objective and Constraint Functions" on page 5-42 contains an example of how to vectorize both for the solver patternsearch. The syntax is nearly identical for ga. The only difference is that patternsearch can have its patterns appear as either row or column vectors; the corresponding vectors for ga are the population vectors, which are always rows.

Parallel Computing with the Genetic Algorithm

In this section...
"Parallel Evaluation of Populations" on page 6-61
"How to Use Parallel Computing with ga" on page 6-61
"Implementation of Parallel Genetic Algorithm" on page 6-63
"Parallel Computing Considerations" on page 6-64

Parallel Evaluation of Populations

Parallel computing is the technique of using multiple processors on a single problem. The reason to use parallel computing is to speed computations.

The Genetic Algorithm and Direct Search Toolbox™ solver ga can automatically distribute the evaluation of objective and nonlinear constraint functions associated with a population to multiple processors. ga uses parallel computing under the following conditions:

- You have a license for Parallel Computing Toolbox™ software.
- Parallel computing is enabled with matlabpool, a Parallel Computing Toolbox function.
- The following options are set using gaoptimset:
  - Vectorized is 'off' (default)
  - UseParallel is 'always'

When these conditions hold, the solver computes the objective function and nonlinear constraint values of the individuals in a population in parallel.

Currently, UseParallel cannot be accessed using the Optimization Tool GUI.

How to Use Parallel Computing with ga

- "Using Parallel Computing with Multicore Processors" on page 6-62
- "Using Parallel Computing with a Multiprocessor Network" on page 6-62

Using Parallel Computing with Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a `matlabpool` of up to 4 parallel workers with a basic Parallel Computing Toolbox license.

Suppose you have a dual-core processor, and wish to use parallel computing:

- Enter

    ```
    matlabpool open 2
    ```

    at the command line. The `2` specifies the number of processors to use.

- Enter

    ```
    options = gaoptimset('UseParallel','always');
    ```

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `'never'`. To halt all parallel computation, enter

```
matlabpool close
```

Using Parallel Computing with a Multiprocessor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB® Distributed Computing Server™ software to establish parallel computation. Here are the steps to take:

1 Obtain a license for Parallel Computing Toolbox functions and MATLAB Distributed Computing Server software.

2 Configure your system for parallel processing. See the Parallel Computing Toolbox documentation, or *MATLAB Distributed Computing Server System Administrator's Guide*.

In particular, if *network_file_path* is the network path to your objective or constraint functions, enter

```
pctRunOnAll('addpath network_file_path')
```

so the worker processors can access your objective or constraint M-files.

Check whether an M-file is on the path of every worker by entering

```
pctRunOnAll('which filename')
```

If any worker does not have a path to the M-file, it reports

```
filename not found.
```

3 At the command line enter

```
matlabpool open conf
```

or

```
matlabpool open conf n
```

where `conf` is your configuration, and `n` is the number of processors you wish to use.

4 Enter

```
options = gaoptimset('UseParallel','always');
```

Once your parallel computing environment is established, applicable solvers automatically use parallel computing whenever called with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `'never'`. To halt all parallel computation, enter

```
matlabpool close
```

Implementation of Parallel Genetic Algorithm

Population generation is implemented in the `ga` solver by using the Parallel Computing Toolbox function `parfor`. `parfor` distributes the evaluation of objective and constraint functions among multiple processes or processors.

The limitations on options, listed in "Parallel Evaluation of Populations" on page 6-61, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `Vectorized` determines whether an entire population is evaluated with one function call, instead of having each member of a population evaluated in a loop. If `Vectorized` is `'on'`, it is not possible to distribute the evaluation of the function using `parfor`, since the evaluation is not done in a loop.

More caveats related to `parfor` are listed in the "Limitations" section of the Parallel Computing Toolbox documentation.

Parallel Computing Considerations

The "Improving Performance with Parallel Computing" section of the Optimization Toolbox™ documentation contains information on factors that affect the speed of parallel computations, factors that affect the results of parallel computations, and searching for global optima. Those considerations also apply to parallel computing with pattern search.

Additionally, there are considerations having to do with random numbers that apply to Genetic Algorithm and Direct Search Toolbox functions. Random number sequences in MATLAB® are pseudorandom, determined from a "seed," an initial setting. Parallel computations use seeds that are not necessarily controllable or reproducible. For example, there is a default global setting on each instance of MATLAB that determines the current seed for random sequences.

Parallel population generation gives nonreproducible results. The pseudorandom sequences cannot be guaranteed to be the same on different runs for many reasons:

- Other functions running on a processor may use random numbers, changing the generated sequences for `ga`.

- Different processors have different conditions, so they may have different sequences.

- The mapping of processor to generated or evaluated individuals may change from run to run.

- A hybrid function run after `ga` may have a different initial condition.

`ga` may have a hybrid function that runs after it finishes; see "Using a Hybrid Function" on page 6-49. If you want the hybrid function to take advantage of

parallel computation, you must set its options separately so that `UseParallel` is `'always'`. If the hybrid function is `patternsearch`, there are two other options that must be set so that `patternsearch` runs in parallel:

- `Cache` must be set to `'off'` (default).
- `CompletePoll` must be set to `'on'`.

If the hybrid function is `fmincon`, the following options must be set in order to take advantage of parallel gradient estimation:

- The option `GradObj` must not be set to `'on'` — it can be `'off'` or `[]`.
- Or, if there is a nonlinear constraint function, the option `GradConstr` must not be set to `'on'` — it can be `'off'` or `[]`.

To find out how to write options for the hybrid function, see "Using a Hybrid Function" on page 6-49 or "Hybrid Function Options" on page 9-40.

7

Using the Simulated Annealing and Threshold Acceptance Algorithms

Using the Simulated Annealing and Threshold Acceptance Algorithms from the Command Line

In this section...
"Running simulannealbnd and threshacceptbnd with the Default Options" on page 7-2
"Setting Options for simulannealbnd and threshacceptbnd at the Command Line" on page 7-3
"Reproducing Your Results" on page 7-5

Running simulannealbnd and threshacceptbnd with the Default Options

To run either the simulated annealing or threshold acceptance algorithms with the default options, call the corresponding function with the syntax

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

The input arguments to simulannealbnd and threshacceptbnd are

- @objfun — A function handle to the M-file that computes the objective function. "Writing M-Files for Functions You Want to Optimize" on page 1-3 explains how to write this M-file.

- x0 — The initial guess of the optimal argument to the objective function.

The output arguments are

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

For a description of additional input and output arguments, see the reference pages for simulannealbnd and threshacceptbnd.

You can run the example described in "Example: Minimizing De Jong's Fifth Function" on page 4-8 from the command line with the simulated annealing algorithm by entering

```
[x,fval] = simulannealbnd(@dejong5fcn, [0 0])
```

This returns

```
x =
   -31.9564  -15.9755
fval =
     5.9288
```

Additional Output Arguments

To get more information about the performance of the algorithm, you can call
`simulannealbnd` or `threshacceptbnd` with the syntax

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

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

- `exitflag` — Flag indicating the reason the algorithm terminated
- `output` — Structure containing information about the performance of the
  algorithm

See the `simulannealbnd` and `threshacceptbnd` reference pages for more
information about these arguments.

Setting Options for simulannealbnd and threshacceptbnd at the Command Line

You can specify options by passing an options structure as an input argument
to either `simulannealbnd` and `threshacceptbnd` using the syntax

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

This syntax does not specify any lower or upper bound constraints.

You create the options structure using the function `saoptimset`:

```
options = saoptimset('simulannealbnd')
```

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

```
options =
          AnnealingFcn: @annealingfast
        TemperatureFcn: @temperatureexp
         AcceptanceFcn: @acceptancesa
                TolFun: 1.0000e-006
         StallIterLimit: '500*numberofvariables'
            MaxFunEvals: '3000*numberofvariables'
              TimeLimit: Inf
                MaxIter: Inf
         ObjectiveLimit: -Inf
                Display: 'final'
        DisplayInterval: 10
              HybridFcn: []
         HybridInterval: 'end'
               PlotFcns: []
           PlotInterval: 1
             OutputFcns: []
     InitialTemperature: 100
       ReannealInterval: 100
               DataType: 'double'
```

These are the default values for `simulannealbnd`. To see the default values for `threshacceptbnd`, run

```
options = saoptimset('threshacceptbnd')
```

The value of each option is stored in a field of the options structure, such as `options.ReannealInterval`. You can display any of these values by entering `options` followed by the name of the field. For example, to display the interval for reannealing used for the simulated annealing algorithm, enter

```
options.ReannealInterval
ans =
    100
```

To create an options structure with a field value that is different from the default—for example, to set `ReannealInterval` to 300 instead of its default value 100—enter

```
options = saoptimset('ReannealInterval', 300)
```

This creates the options structure with all values set to their defaults, except for `ReannealInterval`, which is set to 300.

If you now enter

```
simulannealbnd(@objfun,x0,[],[],options)
```

`simulannealbnd` runs the simulated annealing algorithm with a reannealing interval of 300.

If you subsequently decide to change another field in the options structure, such as setting `PlotFcns` to `@saplotbestf`, which plots the best objective function value at each iteration, call `saoptimset` with the syntax

```
options = saoptimset(options,'PlotFcns',@saplotbestf)
```

This preserves the current values of all fields of `options` except for `PlotFcns`, which is changed to `@saplotbestf`. Note that if you omit the input argument `options`, `saoptimset` resets `ReannealInterval` to its default value 100.

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

```
options = saoptimset('ReannealInterval',300, ...
                     'PlotFcns',@saplotbestf)
```

Reproducing Your Results

Because the simulated annealing and threshold acceptance algorithms are stochastic—that is, they each makes random choices—you get slightly different results each time you run them. The algorithms use the MATLAB® uniform and normal random number generators, `rand` and `randn`, when generating subsequent points and also when determining whether or not to accept new points. Each time the algorithms call `rand` and `randn`, their states are changed so that the next time they are called, they return different random numbers.

If you need to reproduce your results exactly, call `simulannealbnd` or `threshacceptbnd` with an output argument that contains the current states of `rand` and `randn` and then reset the states to these values before running the function again. For example, to reproduce the output of `simulanneal` applied to De Jong's fifth function, call `simulannealbnd` with the syntax

```
[x,fval,exitflag,output] = simulannealbnd(@dejong5fcn,[0 0]);
```

Suppose the results are

```
x =
  -32.0401  -16.1223
fval =
    5.9288
```

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

```
output =
      iterations: 2041
       funccount: 2058
         message: [1x80 char]
       randstate: [625x1 uint32]
      randnstate: [2x1 double]
     problemtype: 'unconstrained'
     temperature: [2x1 double]
       totaltime: 1.8226
```

Reset the states by entering

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

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

You can reproduce your run in the Optimization Tool by checking the box **Use random states from previous run** in the **Run solver and view results** section.
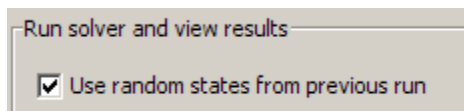
Note If you do not need to reproduce your results, it is better not to set the states of `rand` and `randn`, so that you get the benefit of the randomness in these algorithms.

Parallel Computing with Simulated Annealing and Threshold Acceptance Algorithms

The simulated annealing and threshold acceptance algorithms do not run in parallel automatically. However, they can call hybrid functions that take advantage of parallel computing. For information on how to set hybrid function options, see "Hybrid Function Options" on page 9-50.

`patternsearch` can be used as a hybrid function that uses parallel computation. You must set its options properly in order for it to compute in parallel. For information on the options to set, see "Parallel Computing with Pattern Search" on page 5-48.

`fmincon` can be used as a hybrid function that uses parallel computation for estimating derivatives by parallel finite differences. You must set its options properly in order for it to compute in parallel. For information on the options to set, see "Parallel Computing for Optimization" in the *Optimization Toolbox™ User's Guide*.

Simulated Annealing and Threshold Acceptance Examples

If you are viewing this documentation in the Help browser, click the following link to see the demo Minimization Using Simulated Annealing And Threshold Acceptance Algorithms. Or, from the MATLAB® command line, type `showdemo('saobjective')`.

8

Multiobjective Optimization

What Is Multiobjective Optimization?

Introduction

You might need to formulate problems with more than one objective, since a single objective with several constraints may not adequately represent the problem being faced. If so, there is a vector of objectives,

$$F(x) = [F_1(x), F_2(x),...,F_m(x)],$$

that must be traded off in some way. The relative importance of these objectives is not generally known until the system's best capabilities are determined and tradeoffs between the objectives fully understood. As the number of objectives increases, tradeoffs are likely to become complex and less easily quantified. The designer must rely on his or her intuition and ability to express preferences throughout the optimization cycle. Thus, requirements for a multiobjective design strategy must enable a natural problem formulation to be expressed, and be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

Multiobjective optimization is concerned with the minimization of a vector of objectives $F(x)$ that can be the subject of a number of constraints or bounds:

$$\min_{x \in \mathrm{R}^n} F(x), \text{ subject to}$$
$$G_i(x) = 0, \ i = 1,...,k_e; \ G_i(x) \le 0, \ i = k_e + 1,...,k; \ l \le x \le u.$$

Note that because $F(x)$ is a vector, if any of the components of $F(x)$ are competing, there is no unique solution to this problem. Instead, the concept of noninferiority [4] (also called Pareto optimality [1] and [2]) must be used to characterize the objectives. A noninferior solution is one in which an improvement in one objective requires a degradation of another. To define this concept more precisely, consider a feasible region, Ω, in the parameter space. x is an element of the n-dimensional real numbers $x \in \mathrm{R}^n$ that satisfies all the constraints, i.e.,

$$\Omega = \left\{ x \in \mathrm{R}^n \right\},$$

subject to

$$G_i(x) = 0, \ i = 1,...,k_e,$$
$$G_i(x) \leq 0, \ i = k_e + 1,...,k,$$
$$l \leq x \leq u.$$

This allows definition of the corresponding feasible region for the objective function space Λ:

$$\Lambda = \left\{ y \in \mathrm{R}^m : y = F(x), x \in \Omega \right\}.$$

The performance vector $F(x)$ maps parameter space into objective function space, as represented in two dimensions in the figure Mapping from Parameter Space into Objective Function Space on page 8-3.
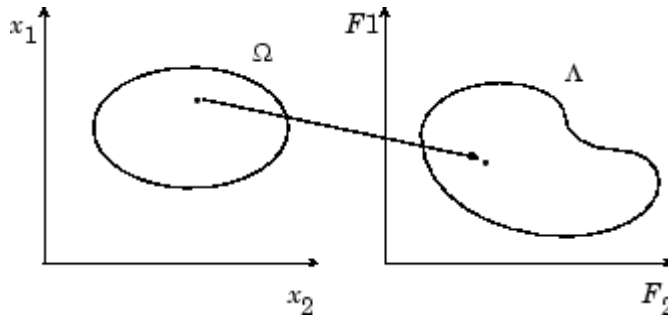
Figure 8-1:   Mapping from Parameter Space into Objective Function Space

A noninferior solution point can now be defined.

Definition: Point $x^* \in \Omega$ is a noninferior solution if for some neighborhood of

x^* there does not exist a Δx such that $(x^* + \Delta x) \in \Omega$ and

$$F_i(x^* + \Delta x) \leq F_i(x^*), \ i = 1,...,m, \ \text{and}$$
$$F_j(x^* + \Delta x) < F_j(x^*) \text{ for at least one } j.$$

In the two-dimensional representation of the figure Set of Noninferior Solutions on page 8-4, the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.
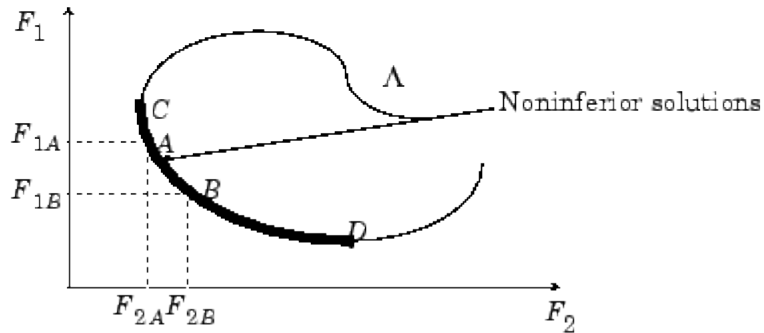
Figure 8-2: Set of Noninferior Solutions

A and *B* are clearly noninferior solution points because an improvement
in one objective, F_1, requires a degradation in the other objective, F_2, i.e.,
$F_{1B} < F_{1A}$, $F_{2B} > F_{2A}$.

Since any point in Ω that is an inferior point represents a point in which
improvement can be attained in all the objectives, it is clear that such a point
is of no value. Multiobjective optimization is, therefore, concerned with the
generation and selection of noninferior solution points.

Noninferior solutions are also called *Pareto optima*. A general goal in
multiobjective optimization is constructing the Pareto optima. The algorithm
used in gamultiobj is described in [3].

Using gamultiobj

Problem Formulation

The gamultiobj solver attempts to create a set of Pareto optima for a multiobjective minimization. You may optionally set bounds and linear constraints on variables. gamultiobj uses the genetic algorithm for finding local Pareto optima. As in the ga function, you may specify an initial population, or have the solver generate one automatically.

The fitness function for use in gamultiobj should return a vector of type double. The population may be of type double, a bit string vector, or can be a custom-typed vector. As in ga, 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)

You can set the initial population in a variety of ways. Suppose that you choose a population of size m. (The default population size is 15 times the number of variables n.) You can set the population:

- As an m-by-n matrix, where the rows represent m individuals.

- As a k-by-n matrix, where $k < m$. The remaining $m - k$ individuals are generated by a creation function.

- The entire population can be created by a creation function.
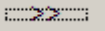
Using gamultiobj with Optimization Tool

You can access gamultiobj from the Optimization Tool GUI. Enter

```
optimtool('gamultiobj')
```

at the command line, or enter optimtool and then choose gamultiobj from
the **Solver** menu. You can also launch the tool from the MATLAB® **Start**
menu as pictured:
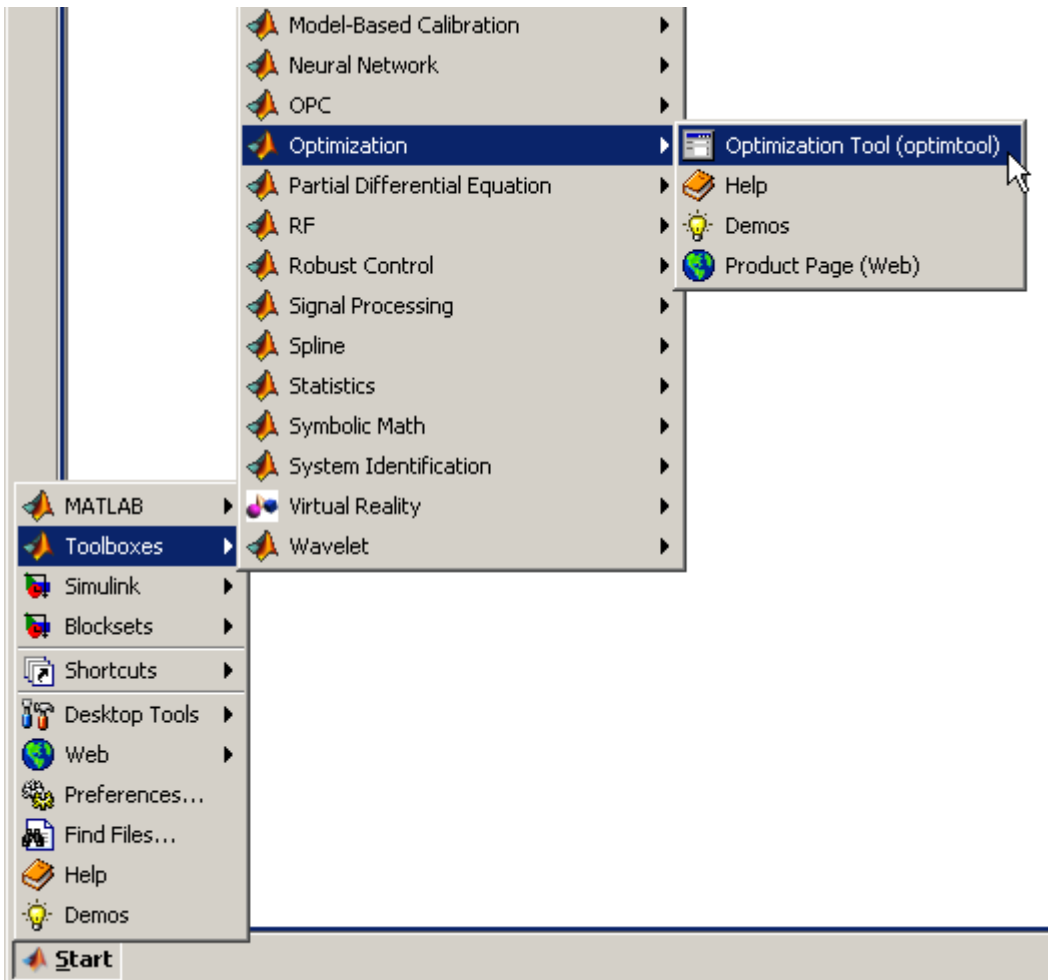
If the **Quick Reference** help pane is closed, you can open it by clicking the ">>" button on the upper right of the GUI: [≡>>≡]. All the options available are explained briefly in the help pane.

You can create an options structure in the Optimization Tool, export it to the MATLAB workspace, and use the structure at the command line. For details, see "Importing and Exporting Your Work" in the Optimization Toolbox™ documentation.

Example — Multiobjective Optimization

This example has a two-objective fitness function $f(x)$, where x is also two-dimensional:
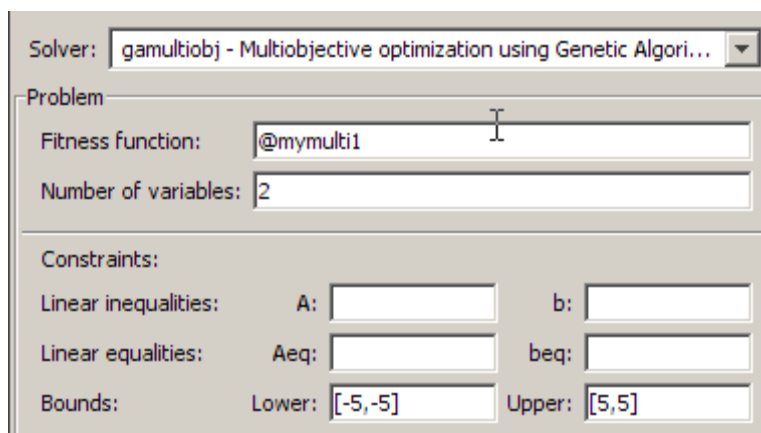
```
function f = mymulti1(x)

f(1) = x(1)^4 - 10*x(1)^2+x(1)*x(2) + x(2)^4 - (x(1)^2)*(x(2)^2);
f(2) = x(2)^4 - (x(1)^2)*(x(2)^2) + x(1)^4 + x(1)*x(2);
```

Create an M-file for this function before proceeding.

Performing the Optimization with Optimization Tool

1 To define the optimization problem, launch the Optimization Tool, and set it as pictured.

2 Set the options for the problem as pictured.

3 Run the optimization by clicking **Start** under **Run solver and view results**.

A plot appears in a figure window.
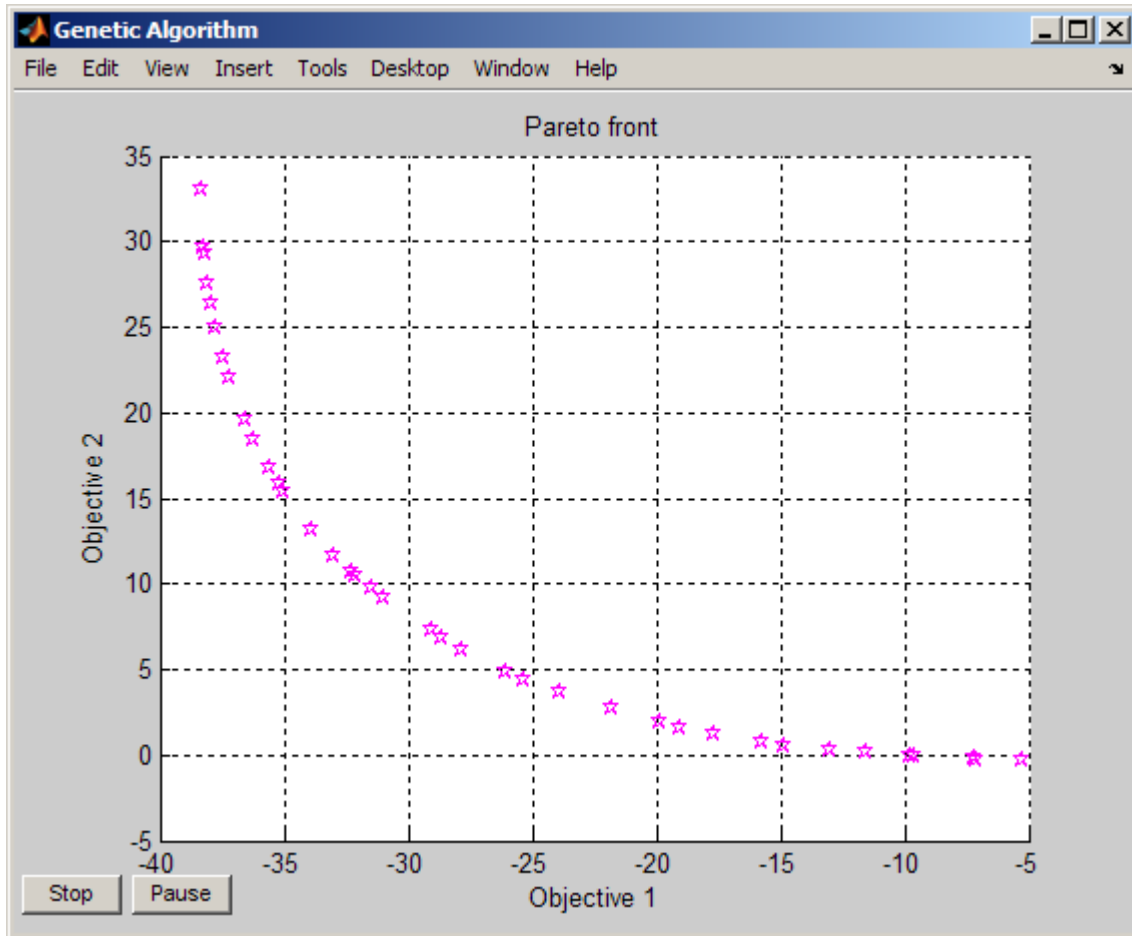
This plot shows the tradeoff between the two components of *f*. It is plotted in objective function space; see the figure Set of Noninferior Solutions on page 8-4.

The results of the optimization appear in the following table containing both objective function values and the value of the variables.

```
-----------------------------
Optimization running.
Optimization terminated.
Optimization terminated: average change in the
spread of Pareto solutions less than
options.TolFun.
```

Pareto front - function values and decision variables

Index	f1	f2	x1 △	x2
1	-5.276	-0.25	0.709	-0.706
2	-38.325	32.745	2.666	-1.951
3	-38.102	27.784	2.567	-1.888
4	-36.608	19.667	2.372	-1.743
5	-35.542	17.129	2.295	-1.833
6	-32.215	10.845	2.075	-1.476
7	-35.147	15.884	2.259	-1.777
8	-20.527	2.142	1.506	-1.19
9	-23.171	3.879	1.645	-1.03
10	-13.605	0.408	1.184	-0.915
11	-19.606	1.897	1.466	-1.076
12	-38.23	29.507	2.603	-1.914
13	-32.625	12.019	2.113	-1.77
14	-38.281	30.518	2.623	-1.932
15	-15.005	0.724	1.254	-0.903
16	-5.276	-0.25	0.709	-0.706
17	-37.953	26.353	2.536	-1.902
18	-33.579	12.685	2.151	-1.583

You can sort the table by clicking a heading. Click the heading again to sort it in the reverse order. The following figures show the result of clicking the heading f1.

Index	f1 △	f2	x1	x2
2	-38.325	32.745	2.666	-1.951
25	-38.314	31.576	2.644	-1.946
14	-38.281	30.518	2.623	-1.932
12	-38.23	29.507	2.603	-1.914
3	-38.102	27.784	2.567	-1.888
17	-37.953	26.353	2.536	-1.902
38	-37.781	25.19	2.509	-1.828
30	-37.517	23.592	2.472	-1.882
28	-37.307	23.55	2.467	-1.717
33	-37.121	21.966	2.431	-1.737
4	-36.608	19.667	2.372	-1.743
27	-36.188	18.75	2.344	-1.848
5	-35.542	17.129	2.295	-1.833
7	-35.147	15.884	2.259	-1.777
26	-34.822	14.928	2.23	-1.675
37	-34.511	14.306	2.209	-1.654
36	-34.14	13.634	2.186	-1.68
18	-33.579	12.685	2.151	-1.583

Pareto front - function values and decision variables

Index	f1 ▽	f2	x1	x2
1	-5.276	-0.25	0.709	-0.706
16	-5.276	-0.25	0.709	-0.706
22	-6.374	-0.196	0.786	-0.871
21	-8.299	-0.164	0.902	-0.874
40	-9.001	-0.119	0.942	-0.909
24	-11.575	0.105	1.081	-0.889
10	-13.605	0.408	1.184	-0.915
15	-15.005	0.724	1.254	-0.903
32	-16.122	0.9	1.305	-1.156
34	-17.394	1.366	1.37	-0.933
11	-19.606	1.897	1.466	-1.076
8	-20.527	2.142	1.506	-1.19
29	-21.411	2.791	1.556	-1.04
19	-22.38	3.014	1.594	-1.145
9	-23.171	3.879	1.645	-1.03
31	-25.141	4.341	1.717	-1.388
35	-26.418	5.115	1.776	-1.367
41	-27.8	6.131	1.842	-1.388

Performing the Optimization at the Command Line

To perform the same optimization at the command line:

1 Set the options:

```
options = gaoptimset('PopulationSize',60,...
          'ParetoFraction',0.7,'PlotFcn',@gaplotpareto);
```

2 Run the optimization using the options:

```
[x fval flag output population] = gamultiobj(@mymulti1,2,...
                          [],[],[],[],[-5,-5],[5,5],options);
```

Alternate Views

There are other ways of regarding the problem. The following figure contains a plot of the level curves of the two objective functions, the Pareto frontier calculated by gamultiobj (boxes), and the x-values of the true Pareto frontier (diamonds connected by a nearly-straight line). The true Pareto frontier points are where the level curves of the objective functions are parallel. They were calculated by finding where the gradients of the objective functions are

parallel. The figure is plotted in parameter space; see the figure Mapping from Parameter Space into Objective Function Space on page 8-3.
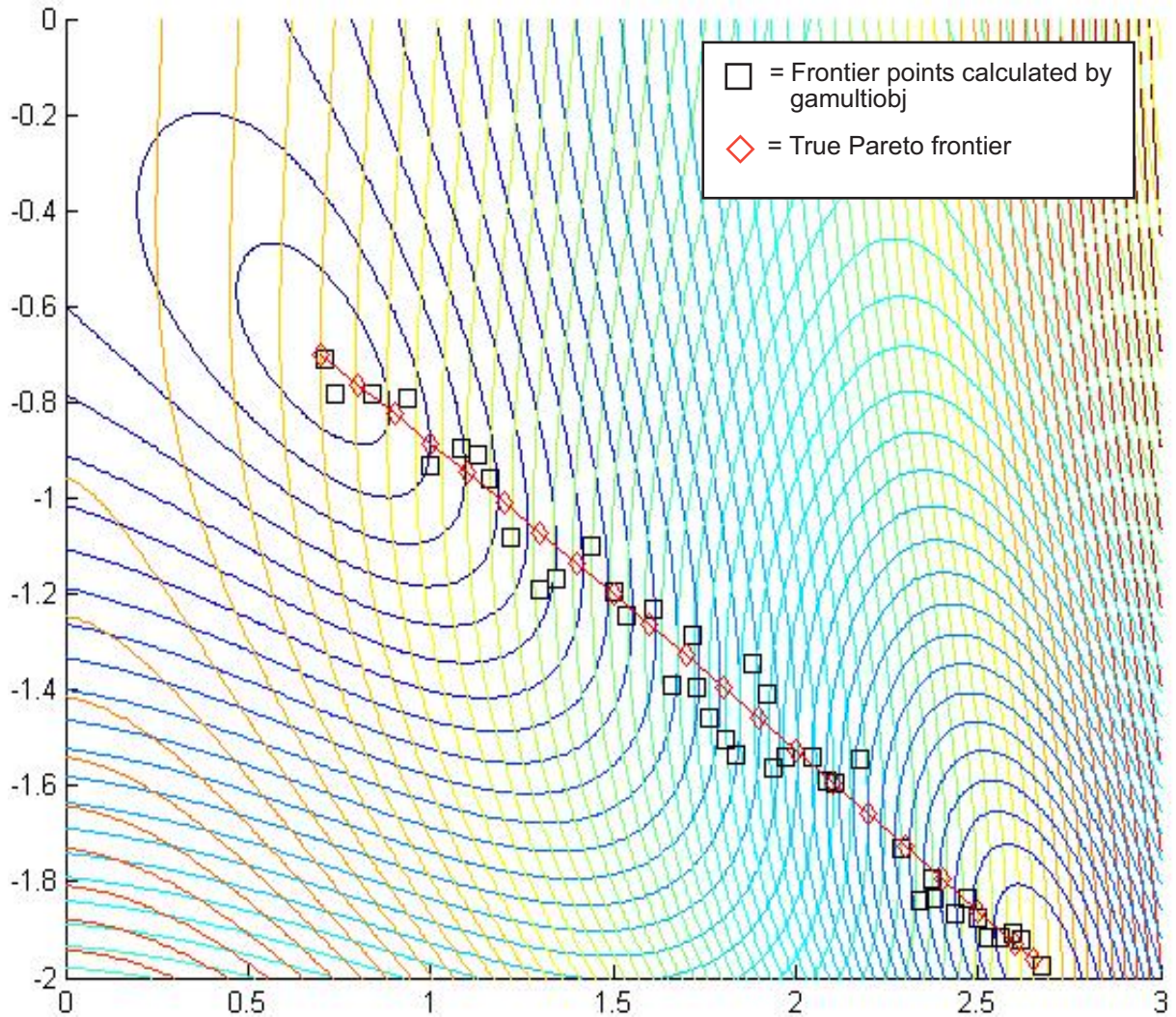
Contours of objective functions, and Pareto frontier

gamultiobj found the ends of the line segment, meaning it found the full extent of the Pareto frontier.

Options and Syntax: Differences With ga

The syntax and options for gamultiobj are similar to those for ga, with the following differences:

- gamultiobj does not have nonlinear constraints, so its syntax has fewer inputs.

- gamultiobj takes an option DistanceMeasureFcn, a function that assigns a distance measure to each individual with respect to its neighbors.

- gamultiobj takes an option ParetoFraction, a number between 0 and 1 that specifies the fraction of the population on the best Pareto frontier to be kept during the optimization. If there is only one Pareto frontier, this option is ignored.

- gamultiobj uses only the Tournament selection function.

- gamultiobj uses elite individuals differently than ga. It sorts noninferior individuals above inferior ones, so it uses elite individuals automatically.

- gamultiobj has only one hybrid function, fgoalattain.

- gamultiobj does not have a stall time limit.

- gamultiobj has different plot functions available.

- gamultiobj does not have a choice of scaling function.

Parallel Computing with gamultiobj

Parallel computing with gamultiobj works almost exactly the same as with ga. For detailed information, see "Parallel Computing with the Genetic Algorithm" on page 6-61.

The difference between parallel computing with gamultiobj and ga has to do with the hybrid function. gamultiobj allows only one hybrid function, fgoalattain. This function is optionally launched after gamultiobj finishes its run. Each individual in the calculated Pareto frontier, i.e., the final population found by gamultiobj, becomes the starting point for an optimization using fgoalattain. These optimizations are done in parallel. The number of processors performing these optimizations is the smaller of the number of individuals and the size of your matlabpool.

For fgoalattain to run in parallel, its options must be set correctly:

```
fgoalopts = optimset('UseParallel','always')
gaoptions = gaoptimset('HybridFcn',{@fgoalattain,fgoalopts});
```

Run gamultiobj with gaoptions, and fgoalattain runs in parallel. For more information about setting the hybrid function, see "Hybrid Function Options" on page 9-40.

gamultiobj calls fgoalattain using a parfor loop, so fgoalattain does not estimate gradients in parallel when used as a hybrid function with gamultiobj. This is because a parfor loop estimates gradients in parallel, and inner iterations in a nested parfor loops do not run in parallel.

References

[1] Censor, Y., "Pareto Optimality in Multiobjective Problems," *Appl. Math. Optimiz.*, Vol. 4, pp 41–59, 1977.

[2] Da Cunha, N.O. and E. Polak, "Constrained Minimization Under Vector-Valued Criteria in Finite Dimensional Spaces," *J. Math. Anal. Appl.*, Vol. 19, pp 103–124, 1967.

[3] Deb, Kalyanmoy, "Multi-Objective Optimization using Evolutionary Algorithms," John Wiley & Sons, Ltd, Chichester, England, 2001.

[4] Zadeh, L.A., "Optimality and Nonscalar-Valued Performance Criteria," *IEEE Trans. Automat. Contr.*, Vol. AC-8, p. 1, 1963.

9

Options Reference

Pattern Search Options

In this section...

Optimization Tool vs. Command Line

There are two ways to specify options for pattern search, depending on whether you are using the Optimization Tool or calling the function patternsearch at the command line:

- If you are using the Optimization Tool, 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.

- 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 5-13 for examples.

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

- By its label, as it appears in the Optimization Tool
- By its field name in the options structure

For example:

- **Poll method** refers to the label of the option in the Optimization Tool.
- PollMethod refers to the corresponding field of the options structure.

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 **Plot functions** 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.
- **Max constraint** (@psplotmaxconstr) plots the maximum nonlinear constraint violation.
- **Custom** enables you to use your own plot function. To specify the plot function using the Optimization 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 9-4 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 function handles to the 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

- nonlinineq — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified

- nonlineq — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified

- flag — Current state in which the plot function is called. The possible values for flag are

  - init — Initialization state

  - iter — Iteration state

  - interrupt — Intermediate stage

  - done — Final state

"Passing Extra Parameters" in the *Optimization Toolbox™ User's Guide* 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 for each of the two classes of direct search algorithms: the generalized pattern search (GPS) algorithm and the mesh adaptive direct search (MADS) algorithm.

These patterns are the Positive basis 2*N* and the Positive basis *N*+1:

- The default pattern, GPS Positive basis 2N, consists of the following 2*N* vectors, where *N* is the number of independent variables for the objective function.

[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[–1 0 0...0]
[0 –1 0...0]
[0 0 0...–1].

For example, if the optimization problem has three independent variables, the pattern consists of the following six vectors.

[1 0 0]
[0 1 0]
[0 0 1]
[–1 0 0]
[0 –1 0]
[0 0 –1].

- The pattern, MADS Positive basis 2N, consists of $2N$ randomly generated vectors, where N is the number of independent variables for the objective function. This is done by randomly generating N vectors which form a linearly independent set, then using this first set and the negative of this set gives $2N$ vectors. As shown above, the GPS Positive basis 2N pattern is formed using the positive and negative of the linearly independent identity, however, with the MADS Positive basis 2N, the pattern is generated using a random permutation of an N-by-N linearly independent lower triangular matrix that is regenerated at each iteration.

- The GPS Positive basis NP1 pattern consists of the following $N + 1$ vectors.

[1 0 0...0]
[0 1 0...0]
...
[0 0 0...1]
[–1 –1 –1...–1].

For example, if the objective function has three independent variables, the pattern consists of the following four vectors.

       [1 0 0]
       [0 1 0]
       [0 0 1]
       [–1 –1 –1].

- The pattern, MADS Positive basis N+1, consists of N randomly generated vectors to form the positive basis, where N is the number of independent variables for the objective function. Then, one more random vector is generated, giving N+1 randomly generated vectors. Each iteration generates a new pattern when the MADS Positive basis N+1 is selected.

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" on page 5-17.

See "Poll Options" on page 9-5 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. Please note, if you have selected the same **Search method** and **Poll method**, only the option selected in the Poll method will be used, although both will be used when the options selected are different.

Complete search (CompleteSearch) applies when you set **Search method** to GPS Positive basis Np1, GPS Positive basis 2N, MADS Positive basis Np1, MADS Positive basis 2N, or Latin hypercube. **Complete search** can have the values On or Off.

For GPS Positive basis Np1, MADS Positive basis Np1, GPS Positive basis 2N, or MADS Positive basis 2N, **Complete search** has the same meaning as the poll option **Complete poll**.

Search method (SearchMethod) specifies the optional search step. The options are

- None ([]) (the default) specifies no search step.
- GPS Positive basis Np1 ('GPSPositiveBasisNp1') performs a search step of a pattern search using the GPS Positive Basis Np1 option.
- GPS Positive basis 2N ('GPSPositiveBasis2N') performs a search step of a pattern search using the GPS Positive Basis 2N option.
- MADS Positive basis Np1 ('MADSPositiveBasisNp1') performs a search step of a pattern search using the MADS Positive Basis Np1option.
- MADS Positive basis 2N ('MADSPositiveBasis2N') performs a search step of a pattern search using the MADS Positive Basis 2N option.
- 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. The default for **Iteration limit** is 1.
  - **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 default), 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. The default for **Iteration limit** is 1.

  - **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. The default for **Design level** is 15.

  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. The default for **Iteration limit** is 1.

- **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 Optimization 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,xBest,fBest,funccount] =
searchfcntemplate(fun,x,A,b,Aeq,beq,lb,ub, ...
    optimValues,options)
```

The search function has the following input arguments:

- fun — Objective function

- x — Current point

- A,b — Linear inequality constraints

- Aeq,beq — Linear equality constraints

- lb,ub — Lower and upper bound constraints

- optimValues — Structure that enables you to set search options. The structure contains the following fields:

  - x — Current point

  - fval — Objective function value at x

  - iteration — Current iteration number

  - funccount — Counter for user function evaluation

  - scale — Scale factor used to scale the design points

  - problemtype — Flag passed to the search routines, indicating whether the problem is 'unconstrained', 'boundconstraints', or 'linearconstraints'. This field is a subproblem type for nonlinear constrained problems.

  - meshsize — Current mesh size used in search step

  - method — Method used in last iteration

- options — Pattern search options structure

The function has the following output arguments:

- successSearch — A Boolean identifier indicating whether the search is successful or not

- xBest,fBest — Best point and best function value found by search method

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, because performing these searches more than once is not likely to improve results.

- funccount — Number of user function evaluation in search method

See "Using a Search Method" on page 5-23 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, and is only used when the GPS algorithm is selected as the Poll or Search method. 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) is only applied when **Poll method** is set to GPS Positive basis Np1. It specifies whether the mesh vectors are multiplied by –1 when the mesh size is less than 1/100 of the mesh tolerance (minimum mesh size TolMesh) after an unsuccessful poll. In other words, after the first unsuccessful poll with small mesh size, instead of polling in directions e_i (unit positive directions) and $-\Sigma e_i$, the algorithm polls in directions $-e_i$ and Σe_i. **Rotate** can have the values Off or On (the default). When the problem has equality constraints, **Rotate** is disabled.

Rotate is especially useful for discontinuous functions.

Note Changing the setting of **Rotate** has no effect on the poll when **Poll method** is set to GPS Positive basis 2N, MADS Positive basis 2N, or MADS Positive basis Np1.

Scale (`ScaleMesh`) specifies whether the algorithm scales the mesh points by carefully multiplying the pattern vectors by constants proportional to the logarithms of the absolute values of components of the current point (or, for unconstrained problems, of the initial point). **Scale** can have the values `Off` or `On` (the default). When the problem has equality constraints, **Scale** is disabled.

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 and is only used when a GPS method is selected as the Poll or Search method.

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 and is only used when a GPS method is selected as the Poll or Search method.

See "Mesh Expansion and Contraction" on page 5-26 for more information.

Algorithm Settings

Algorithm settings define algorithmic specific parameters.

Parameters that can be specified for a nonlinear constraint algorithm include

- **Initial penalty** (`InitialPenalty`) — Specifies an initial value of the penalty parameter that is used by the algorithm. **Initial penalty** must be greater than or equal to 1.
- **Penalty factor** (`PenaltyFactor`) — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. **Penalty factor** must be greater than 1.

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 5-32 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 objective 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

Time limit (`TimeLimit`) specifies the maximum time in seconds the pattern search algorithm runs before stopping. This also includes any specified pause time for pattern search algorithms.

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. After a successful poll, if the difference between the function value at the previous best point and function value at the current best point is less than the value of Function tolerance, the algorithm halts. The default value is `1e-6`.

See "Setting Tolerances for the Solver" on page 5-34 for an example.

Nonlinear constraint tolerance (`TolCon`) — The **Nonlinear constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints.

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 Optimization 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
  ```

  at the MATLAB command prompt.

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

- nonlinineq — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified

- nonlineq — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified

- 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

  - interrupt — Intermediate stage

  - done — Final state

- interval — Optional interval argument

"Passing Extra Parameters" in the *Optimization Toolbox User's Guide* 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') — No output is displayed.

- Iterative ('iter') — Information is displayed for each iteration.

- Diagnose ('diagnose') — Information is displayed for each iteration. In addition, the diagnostic lists some problem information and the options that are changed from the defaults.

- Final ('final') — The reason for stopping is displayed.

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 (with no nonlinear constraint function specified). With a nonlinear constraint function, Method displays the update method used after a subproblem is solved.
- Max Constraint — Maximum nonlinear constraint violation (displayed only when a nonlinear constraint function has been specified)

The default value of **Level of display** is

- Off in the Optimization Tool
- 'final' in an options structure created using psoptimset

Vectorize Option

Objective function is vectorized specifies whether the computation of the objective function is vectorized. When **Objective function is vectorized** is Off, the algorithm calls the objective function on one point at a time as it loops through all of the mesh points. When **Objective function is vectorized** is On, the pattern search algorithm calls the objective function on all the points in the mesh at once, i.e., in a single call to the objective function if either **Complete Poll** or **Complete Search** is On.

If there are nonlinear constraints, the objective function and the nonlinear constraints all need to be vectorized in order for the algorithm to compute in a vectorized manner.

Parallel Option

Specifies whether patternsearch computes applicable polls and searches in parallel, using multiple processes or processors, or computes them serially. Set UseParallel to 'always' to compute in parallel. Set UseParallel to 'never' to compute serially.

Options Table for Pattern Search Algorithms

Option Availability Table for GPS and MADS Algorithms

Option	Description	Algorithm Availability
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. If the objective function is stochastic, it is advised not to use this option.	GPS, MADS
CacheSize	Size of the cache, in number of points.	GPS, MADS
CacheTol	Positive scalar specifying how close the current mesh point must be to a point in the cache in order for patternsearch to avoid polling it. Available if 'Cache' option is set to 'on'.	GPS, MADS

Option Availability Table for GPS and MADS Algorithms (Continued)

Option	Description	Algorithm Availability
CompletePoll	Complete poll around current iterate. Evaluate all the points in a poll step.	GPS, MADS
CompleteSearch	Complete poll around current iterate. Evaluate all the points in a search step.	GPS, MADS
Display	Level of display to Command Window.	GPS, MADS
InitialMeshSize	Initial mesh size used in pattern search algorithms.	GPS, MADS
InitialPenalty	Initial value of the penalty parameter.	GPS, MADS
MaxFunEvals	Maximum number of objective function evaluations.	GPS, MADS
MaxIter	Maximum number of iterations.	GPS, MADS
MaxMeshSize	Maximum mesh size used in a poll/search step.	GPS, MADS
MeshAccelerator	Accelerate mesh size contraction.	GPS, MADS
MeshContraction	Mesh contraction factor, used when iteration is unsuccessful.	GPS — Default value is 0.5. MADS — Default value is 0.25 if MADS algorithm is selected for either the **Poll method** or **Search method**.

Option Availability Table for GPS and MADS Algorithms (Continued)

Option	Description	Algorithm Availability
MeshExpansion	Mesh expansion factor, expands mesh when iteration is successful.	GPS — Default value is 2. MADS — Default value is 4 if MADS algorithm is selected for either the **Poll method** or **Search method**.
MeshRotate	Rotate the pattern before declaring a point to be optimum.	GPS, MADS
OutputFcn	User-specified function that a pattern search calls at each iteration.	GPS, MADS
PenaltyFactor	Penalty update parameter.	GPS, MADS
PlotFcn	Specifies function to plot at runtime.	GPS, MADS
PlotInterval	Specifies that plot functions will be called at every interval.	GPS, MADS
PollingOrder	Order in which search directions are polled.	GPS only
PollMethod	Polling strategy used in pattern search.	GPS, MADS
ScaleMesh	Automatic scaling of variables.	GPS, MADS
SearchMethod	Specifies search method used in pattern search.	GPS, MADS

Option Availability Table for GPS and MADS Algorithms (Continued)

Option	Description	Algorithm Availability
TimeLimit	Total time (in seconds) allowed for optimization. Also includes any specified pause time for pattern search algorithms.	GPS, MADS
TolBind	Binding tolerance used to determine if linear constraint is active.	GPS, MADS
TolCon	Tolerance on nonlinear constraints.	GPS, MADS
TolFun	Tolerance on function value.	GPS, MADS
TolMesh	Tolerance on mesh size.	GPS, MADS
TolX	Tolerance on independent variable.	GPS, MADS
UseParallel	When 'always', compute objective functions of a poll or search in parallel. Disable by setting to 'never'.	GPS, MADS
Vectorized	Specifies whether functions are vectorized.	GPS, MADS

Genetic Algorithm Options

In this section...

Optimization Tool vs. Command Line

There are two ways to specify options for the genetic algorithm, depending on whether you are using the Optimization Tool or calling the functions ga or at the command line:

- If you are using the Optimization Tool (`optimtool`), select an option from a drop-down list or enter the value of the option in a text field.

- If you are calling ga or `gamultiobj` from the command line, create 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 6-13 for examples.

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

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

For example:

- **Population type** is the label of the option in the Optimization Tool.
- `PopulationType` is the corresponding field of the `options` structure.

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. Click on any subplot to view a larger version of the plot in a separate figure window. 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 **Plot functions** 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.

- **Max constraint** (@gaplotmaxconstr) plots the maximum nonlinear constraint violation 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 Optimization 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 9-26.

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 function handles to the 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 9-26 describes the fields of `state`.

- `flag` — String that tells what stage the algorithm is currently in.

"Passing Extra Parameters" in the *Optimization Toolbox™ User's Guide* 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 genetic algorithm 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

- `NonlinIneq` — Nonlinear inequality constraints, displayed only when a nonlinear constraint function is specified

- NonlinEq — Nonlinear equality constraints, displayed only when a nonlinear constraint function is specified

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 if there are no constraints or bound constraints.

- Feasible population (@gacreationlinearfeasible) creates a random initial population that satisfies all bounds and linear constraints. It is biased to create individuals that are on the boundaries of the constraints, and to create well-dispersed populations. This is the default if there are linear constraints.

- 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 Optimization 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.

  "Passing Extra Parameters" in the *Optimization Toolbox User's Guide* 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 default **Creation function** to create an initial population. If you enter a

nonempty array in the **Initial population** field, the array must have no more than **Population size** rows, and exactly **Number of variables** columns. In this case, the genetic algorithm calls a **Creation function** to generate the remaining individuals, if required.

Initial scores (`InitialScores`) specifies initial scores for the initial population. The initial scores can also be partial.

Initial range (`PopInitRange`) specifies the range of the vectors in the initial population that is generated by a 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 6-23 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 Optimization 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 expectation = 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.

"Passing Extra Parameters" in the *Optimization Toolbox User's Guide* explains how to provide additional parameters to the function.

See "Fitness Scaling" on page 6-31 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 Optimization 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.

  "Passing Extra Parameters" in the *Optimization Toolbox User's Guide* explains how to provide additional parameters to the function.

See "Selection" on page 6-34 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 6-38 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 standard deviation 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 standard deviation at the first generation. If you set **Initial range** to be a 2-by-1 vector v, the initial standard deviation 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 standard deviation at coordinate i of the parent vector is given by **Scale**$*($v$($i,2) -v($i,1$))$.

- The **Shrink** parameter controls how the standard deviation shrinks as generations go by. If you set **Initial range** to be a 2-by-1 vector, the standard deviation at the kth generation, σ_k, is the same at all coordinates of the parent vector, and is given by the recursive formula

$$\sigma_k = \sigma_{k-1}\left(1 - \text{Shrink}\,\frac{k}{\text{Generations}}\right).$$

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

$$\sigma_{i,k} = \sigma_{i,k-1}\left(1 - \text{Shrink}\,\frac{k}{\text{Generations}}\right).$$

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

The default value of both **Scale** and **Shrink** is 1. To change the 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**.

- Adaptive Feasible (mutationadaptfeasible) randomly generates directions that are adaptive with respect to the last successful or unsuccessful generation. The feasible region is bounded by the constraints and inequality constraints. A step length is chosen along each direction so that linear constraints and bounds are satisfied.

- Custom enables you to write your own mutation function. To specify the mutation function using the Optimization 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 9-26 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**.

"Passing Extra Parameters" in the *Optimization Toolbox User's Guide* 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**.

- Arithmetic (@crossoverarithmetic) creates children that are the weighted arithmetic mean of two parents. Children are always feasible with respect to linear constraints and bounds.

- Custom enables you to write your own crossover function. To specify the crossover function using the Optimization 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 — Placeholder 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**.

"Passing Extra Parameters" in the *Optimization Toolbox User's Guide* 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.

- **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.

Algorithm Settings

Algorithm settings define algorithmic specific parameters.

Parameters that can be specified for a nonlinear constraint algorithm include

- **Initial penalty** (InitialPenalty) — Specifies an initial value of the penalty parameter that is used by the algorithm. **Initial penalty** must be greater than or equal to 1.

- **Penalty factor** (PenaltyFactor) — Increases the penalty parameter when the problem is not solved to required accuracy and constraints are not satisfied. **Penalty factor** must be greater than 1.

Multiobjective Options

Multiobjective options define parameters characteristic of the multiobjective genetic algorithm. You can specify the following parameters:

- DistanceMeasureFcn — Defines a handle to the function that computes distance measure of individuals, computed in decision variable or design space (genotype) or in function space (phenotype). For example, the default distance measure function is distancecrowding in function space, or {@distancecrowding,'phenotype'}.

- ParetoFraction — Sets the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts. This option is a scalar between 0 and 1.

Hybrid Function Options

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 to perform unconstrained minimization.

- patternsearch (@patternsearch) — Uses a pattern search to perform constrained or unconstrained minimization.

- fminunc (@fminunc) — Uses the Optimization Toolbox function fminunc to perform unconstrained minimization.

- fmincon (@fmincon) — Uses the Optimization Toolbox function fmincon to perform constrained minimization.

You can set a separate options structure for the hybrid function. Use psoptimset or optimset to create the structure, depending on whether the hybrid function is patternsearch or not:

```
hybridopts = optimset('display','iter','LargeScale','off');
```

Include the hybrid options in the Genetic Algorithm options structure as follows:

```
options = gaoptimset(options,'HybridFcn',{@fminunc,hybridopts});
```

hybridopts must exist before you set options.

See "Using a Hybrid Function" on page 6-49 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 for the genetic algorithm to 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 the weighted average change in the fitness function value over **Stall generations** is less than **Function tolerance**.

- **Stall time limit** (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**.

- **Function tolerance** (TolFun) — The algorithm runs until the cumulative change in the fitness function value over **Stall generations** is less than or equal to **Function Tolerance**.

- **Nonlinear constraint tolerance** (TolCon) — The **Nonlinear constraint tolerance** is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints.

See "Setting the Maximum Number of Generations" on page 6-53 for an example.

Output Function Options

Output functions are functions that the genetic algorithm calls at each generation. The following options are available:

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 Optimization 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.

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 9-26 describes the fields of state.

- flag — String indicating the current status of the algorithm as follows:

    - 'init' — Initial stage

    - 'iter' — Algorithm running

    - 'interrupt' — Intermediate stage

    - 'done' — Algorithm terminated

- interval — Optional interval argument

"Passing Extra Parameters" in the *Optimization Toolbox User's Guide* 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') — No output is displayed.

- Iterative ('iter') — Information is displayed at each iteration.

- Diagnose (`'diagnose'`) — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.

- Final (`'final'`) — The reason for stopping is displayed.

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

When a nonlinear constraint function has been specified, `Iterative` and `Diagnose` will not display the `Mean f(x)`, but will additionally display:

- `Max Constraint` — Maximum nonlinear constraint violation

The default value of **Level of display** is

- `Off` in the Optimization Tool
- `'final'` in an options structure created using `gaoptimset`

Vectorize Option

The vectorize option specifies whether the computation of the fitness function is vectorized. Set **Set Objective function is vectorized** to `On` to indicate that the fitness function is vectorized. When **Objective function is vectorized** is `Off`, the algorithm calls the fitness function on one individual at a time as it loops through the population.

See "Vectorizing the Fitness Function" on page 6-54 for an example.

Parallel Option

Specifies whether `ga` or `gamultiobj` evaluates populations in parallel, using multiple processes or processors, or evaluates them serially. Set `UseParallel` to `'always'` to evaluate in parallel. Set `UseParallel` to `'never'` to evaluate serially.

Simulated Annealing and Threshold Acceptance Algorithm Options

In this section...
"saoptimset At The Command Line" on page 9-46
"Plot Options" on page 9-46
"Temperature Options" on page 9-48
"Algorithm Settings" on page 9-49
"Hybrid Function Options" on page 9-50
"Stopping Criteria Options" on page 9-51
"Output Function Options" on page 9-51
"Display Options" on page 9-53

saoptimset At The Command Line

You specify options by creating an `options` structure using the function `saoptimset`, as follows:

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

See "Setting Options for simulannealbnd and threshacceptbnd at the Command Line" on page 7-3 for examples.

Each option in this section is listed by its field name in the `options` structure. For example, `InitialTemperature` refers to the corresponding field of the `options` structure.

Plot Options

Plot options enable you to plot data from the simulated annealing or threshold acceptance solver while it is running. When you specify plot functions and run the algorithm, a plot window displays the plots on separate axes. Right-click on any subplot to view a larger version of the plot in a separate figure window.

PlotInterval specifies the number of iterations between consecutive calls to the plot function.

To display a plot when calling simulannealbnd or threshacceptbnd from the command line, set the PlotFcns field of options to be a function handle to the plot function. You can specify any of the following plots:

- @saplotbestf plots the best objective function value.

- @saplotbestx plots the current best point.

- @saplotf plots the current function value.

- @saplotx plots the current point.

- @saplotstopping plots stopping criteria levels.

- @saplottemperature plots the temperature at each iteration.

- @myfun plots a custom plot function, where myfun is the name of your function. See "Structure of the Plot Functions" on page 9-4 for a description of the syntax.

For example, to display the best objective plot, set options as follows

```
options = saoptimset('PlotFcns',@saplotbestf);
```

To display multiple plots, use the cell array syntax

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

where @plotfun1, @plotfun2, and so on are function handles to the plot functions.

Structure of the Plot Functions

The first line of a plot function has the form

```
function stop = plotfun(options,optimvalues,flag)
```

The input arguments to the function are

- options — Options structure created using saoptimset.

- `optimvalues` — Structure containing information about the current state of the solver. The structure contains the following fields:

  - `x` — Current point

  - `fval` — Objective function value at x

  - `bestx` — Best point found so far

  - `bestfval` — Objective function value at best point

  - `temperature` — Current temperature

  - `iteration` — Current iteration

  - `funccount` — Number of function evaluations

  - `t0` — Start time for algorithm

  - `k` — Annealing parameter

  - `tau` — Threshold acceptance sequence (for threshold acceptance solver only)

- `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

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.

Temperature Options

Temperature options specify how the temperature will be lowered at each iteration over the course of the algorithm.

- `InitialTemperature` — Initial temperature at the start of the algorithm. The default is `100`.

- TemperatureFcn — Function used to update the temperature schedule. Let i denote the iteration number. The options are:

  - @temperatureexp — The temperature is equal to InitialTemperature * 0.95^i. This is the default.

  - @temperaturefast — The temperature is equal to InitialTemperature / i.

  - @temperatureboltz — The temperature is equal to InitialTemperature / ln(i).

  - @myfun — Uses a custom function, myfun, to update temperature. See the functions above for syntax.

- ReannealInterval — Number of points accepted before reannealing. The default value is 100.

Algorithm Settings

Algorithm settings define algorithmic specific parameters used in generating new points at each iteration.

Parameters that can be specified for the simulated annealing and threshold acceptance algorithms are:

- AnnealingFcn — Function used to generate new points for the next iteration. The choices are:

  - @annealingfast — The step has length temperature, with direction uniformly at random. This is the default.

  - @annealingboltz — The step has length square root of temperature, with direction uniformly at random.

  - @myfun — Uses a custom annealing algorithm, myfun. The syntax is:

    ```
    newx = myfun(optimValues,problem)
    ```

    where optimValues is a structure described in "Structure of the Output Function" on page 9-52.

AcceptanceFcn — Function used to determine whether a new point is accepted or not. The default function varies depending on which solver/algorithm you are using. The choices are:

- `@acceptancesa` — Simulated annealing acceptance function. The default for `simulannealbnd`. This function cannot be used with `threshacceptbnd`.

- `@acceptancethresh` — Threshold acceptance function. The default for `threshacceptbnd`. This function cannot be used with `simulannealbnd`.

- `@myfun` — A custom acceptance function, `myfun`. The syntax is:

```
newpoint = myfun(optimValues,newx,newfval);
```

  where `optimValues` is a structure described in "Structure of the Output Function" on page 9-52, `newx` is the point being evaluated for acceptance, and `newfval` is the objective function at `newx`.

Hybrid Function Options

A hybrid function is another minimization function that runs during or at the end of iterations of the solver. `HybridInterval` specifies the interval (if not `never` or `end`) at which the hybrid function is called. You can specify a hybrid function using the `HybridFcn` option. The choices are:

- `[]` — No hybrid function.

- `@fminsearch` — Uses the MATLAB® function `fminsearch` to perform unconstrained minimization.

- `@patternsearch` — Uses `patternsearch` to perform constrained or unconstrained minimization.

- `@fminunc` — Uses the Optimization Toolbox™ function `fminunc` to perform unconstrained minimization.

- `@fmincon` — Uses the Optimization Toolbox function `fmincon` to perform constrained minimization.

You can set a separate options structure for the hybrid function. Use `psoptimset` or `optimset` to create the structure, depending on whether the hybrid function is `patternsearch` or not:

```
hybridopts = optimset('display','iter','LargeScale','off');
```

Include the hybrid options in the Simulated Annealing or Threshold Acceptance Algorithm `options` structure as follows:

```
options = saoptimset(options,'HybridFcn',{@fminunc,hybridopts});
```

hybridopts must exist before you set options.

See "Using a Hybrid Function" on page 6-49 for an example.

Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

- TolFun — The algorithm runs until the average change in value of the objective function in StallIterLim iterations is less than TolFun. The default value is 1e-6.

- MaxIter — The algorithm stops if the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or Inf. Inf is the default.

- MaxFunEval specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the maximum number of function evaluations. The allowed maximum is 3000*numberofvariables.

- TimeLimit specifies the maximum time in seconds the algorithm runs before stopping.

- ObjectiveLimit — The algorithm stops if the best objective function value is less than or equal to the value of ObjectiveLimit.

Output Function Options

Output functions are functions that the algorithm calls at each iteration. The default value is to have no output function, []. You must first create an output function using the syntax described in "Structure of the Output Function" on page 9-52. Then, specify your function as @myfun, where myfun is the name of your function.

If you are using simulanneal, set

```
options = saoptimset('OutputFcns',@myfun);
```

To see a template that you can use to write your own output functions, enter

```
edit saoutputfcntemplate
```

at the MATLAB command line.

Structure of the Output Function

The output function has the following calling syntax.

```
[stop,options,optchanged] = myfun(options,optimvalues,flag)
```

The function has the following input arguments:

- options — Options structure created using saoptimset.
- optimvalues — Structure containing information about the current state of the solver. The structure contains the following fields:
  - x — Current point
  - fval — Objective function value at x
  - bestx — Best point found so far
  - bestfval — Objective function value at best point
  - temperature — Current temperature
  - iteration — Current iteration
  - funccount — Number of function evaluations
  - t0 — Start time for algorithm
  - k — Annealing parameter
  - tau — Threshold acceptance sequence (for threshold acceptance solver only)
- 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

The output function returns the following arguments:

- 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 modified by the output function.

- optchanged — A boolean flag indicating changes were made to options. This must be set to true if options are changed.

Display Options

Use the Display option to specify how much information is displayed at the command line while the algorithm is running. The available options are

- off — No output is displayed. This is the default value for an options structure created using saoptimset.

- iter — Information is displayed at each iteration.

- diagnose — Information is displayed at each iteration. In addition, the diagnostic lists some problem information and the options that have been changed from the defaults.

- final — The reason for stopping is displayed. This is the default.

Both iter and diagnose display the following information:

- Iteration — Iteration number

- f-count — Cumulative number of objective function evaluations

- Best f(x) — Best objective function value

- Current f(x) — Current objective function value

- Mean Temperature — Mean temperature function value

Function Reference

Genetic Algorithm

ga	Find minimum of function using genetic algorithm
gamultiobj	Find minima of multiple functions using genetic algorithm
gaoptimget	Obtain values of genetic algorithm options structure
gaoptimset	Create genetic algorithm options structure

Direct Search

patternsearch	Find minimum of function using pattern search
psoptimget	Obtain values of pattern search options structure
psoptimset	Create pattern search options structure

Simulated Annealing and Threshold Acceptance Algorithms

saoptimget	Values of simulated annealing or threshold acceptance algorithm options structure
saoptimset	Create simulated annealing algorithm or threshold acceptance options structure

`simulannealbnd`	Find unconstrained or bound-constrained minimum of function of several variables using simulated annealing algorithm
`threshacceptbnd`	Find unconstrained or bound-constrained minimum of function of several variables using threshold acceptance algorithm

Functions — Alphabetical List

ga

Purpose	Find minimum of function using genetic algorithm
Syntax	

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

Description ga implements the genetic algorithm at the command line to minimize an objective function.

x = ga(fitnessfcn,nvars) finds a local unconstrained minimum, x, to the objective function, fitnessfcn. nvars is the dimension (number of design variables) of fitnessfcn. The objective function, fitnessfcn, accepts a vector x of size 1-by-nvars, and returns a scalar evaluated at x.

Note To write a function with additional parameters to the independent variables that can be called by ga, see the section on "Passing Extra Parameters" in the *Optimization Toolbox™ User's Guide*.

---

x = ga(fitnessfcn,nvars,A,b) finds a local minimum x to fitnessfcn, subject to the linear inequalities $A*x \le b$. fitnessfcn accepts input x and returns a scalar function value evaluated at x.

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.

Note that the linear constraints are not satisfied when the PopulationType option is set to 'bitString' or 'custom'.

x = ga(fitnessfcn,nvars,A,b,Aeq,beq) finds a local minimum x to fitnessfcn, subject to the linear equalities $Aeq * x = beq$ as well as $A * x \leq b$. (Set A=[] and b=[] if no inequalities exist.)

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.

Note that the linear constraints are not satisfied when the PopulationType option is set to 'bitString' or 'custom'.

x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB) defines a set of lower and upper bounds on the design variables, x, so that a solution is found in the range $LB \leq x \leq UB$. Use empty matrices for LB and UB if no bounds exist. Set *LB*(*i*) = *-Inf* if *x*(*i*) is unbounded below; set *UB*(*i*) = *Inf* if *x*(*i*) is unbounded above.

x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon) subjects the minimization to the constraints defined in nonlcon. The function nonlcon accepts x and returns the vectors C and Ceq, representing the nonlinear inequalities and equalities respectively. ga minimizes the fitnessfcn such that C(x)≤0 and Ceq(x)=0. (Set LB=[] and UB=[] if no bounds exist.)

Note that the nonlinear constraints are not satisfied when the PopulationType option is set to 'bitString' or 'custom'.

x = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options) minimizes with the default optimization parameters replaced by values in the structure options, which can be created using the gaoptimset function. See the gaoptimset reference page for details.

x = ga(problem) finds the minimum for problem, where problem is a structure containing the following fields:

fitnessfcn	Fitness function
nvars	Number of design variables

Aineq	A matrix for linear inequality constraints
Bineq	b vector for linear inequality constraints
Aeq	A matrix for linear equality constraints
Beq	b vector for linear equality constraints
lb	Lower bound on x
ub	Upper bound on x
nonlcon	Nonlinear constraint function
randstate	Optional field to reset rand state
randnstate	Optional field to reset randn state
solver	'ga'
options	Options structure created using gaoptimset

Create the structure problem by exporting a problem from Optimization Tool, as described in "Importing and Exporting Your Work" in the *Optimization Toolbox User's Guide*.

[x,fval] = ga(...) returns fval, the value of the fitness function at x.

[x,fval,exitflag] = ga(...) returns exitflag, an integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated.

- 1 — Average cumulative change in value of the fitness function over options.StallGenLimit generations less than options.TolFun and constraint violation less than options.TolCon.

- 2 — Fitness limit reached and constraint violation less than options.TolCon.

- 3 — The value of the fitness function did not change in options.StallGenLimit generations and constraint violation less than options.TolCon.

- `4` — Magnitude of step smaller than machine precision and constraint violation less than `options.TolCon`.

- `0` — Maximum number of generations exceeded.

- `-1` — Optimization terminated by the output or plot function.

- `-2` — No feasible point found.

- `-4` — Stall time limit exceeded.

- `-5` — Time limit exceeded.

`[x,fval,exitflag,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 6-17.

- `generations` — The number of generations computed.

- `funccount` — The number of evaluations of the fitness function

- `message` — The reason the algorithm terminated.

- `maxconstraint` — Maximum constraint violation, if any.

`[x,fval,exitflag,output,population] = ga(...)` returns the matrix, `population`, whose rows are the final population.

`[x,fval,exitflag,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 accept 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 inequality constraints and lower bounds

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$

$$x_1 \geq 0, \quad x_2 \geq 0,$$

the following code finds the minimum of the function, `lincontest6`, that is provided your software:

```
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
lb = zeros(2,1);
[x,fval,exitflag] = ga(@lincontest6,...
2,A,b,[],[],lb)

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

x =
   0.7794   1.2205

fval =
   -8.03916

exitflag =
1
```

References   [1] Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

[2] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

[3] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds", *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

See Also   `gaoptimget`, `gaoptimset`, `patternsearch`, `simulannealbnd`, `threshacceptbnd`

gamultiobj

Purpose　　　　Find minima of multiple functions using genetic algorithm

Syntax
```
X = gamultiobj(FITNESSFCN,NVARS)
X = gamultiobj(FITNESSFCN,NVARS,A,b)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB)
X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,options)
X = gamultiobj(problem)
[X,FVAL] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG,OUTPUT] = gamultiobj(FITNESSFCN,NVARS, ...)
[X,FVAL,EXITFLAG,OUTPUT,POPULATION] = gamultiobj(FITNESSFCN,
    ...)
[X,FVAL,EXITFLAG,OUTPUT,POPULATION,
    SCORE] = gamultiobj(FITNESSFCN, ...)
```

Description　　gamultiobj implements the genetic algorithm at the command line to minimize a multicomponent objective function.

X = gamultiobj(FITNESSFCN,NVARS) finds a local Pareto set X of the objective functions defined in FITNESSFCN. NVARS is the dimension of the optimization problem (number of decision variables). FITNESSFCN accepts a vector X of size 1-by-NVARS and returns a vector of size 1-by-numberOfObjectives evaluated at a decision variable. X is a matrix with NVARS columns. The number of rows in X is the same as the number of Pareto solutions. All solutions in a Pareto set are equally optimal; it is up to the designer to select a solution in the Pareto set depending on the application.

X = gamultiobj(FITNESSFCN,NVARS,A,b) finds a local Pareto set X of the objective functions defined in FITNESSFCN, subject to the linear inequalities $A*x \leq b$. Linear constraints are supported only for the default PopulationType option ('doubleVector'). Other population types, e.g., 'bitString' and 'custom', are not supported.

X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq) finds a local Pareto set X of the objective functions defined in FITNESSFCN, subject

to the linear equalities $Aeq * x = beq$ as well as the linear inequalities $A * x \leq b$. (Set A=[] and b=[] if no inequalities exist.) Linear constraints are supported only for the default PopulationType option ('doubleVector'). Other population types, e.g., 'bitString' and 'custom', are not supported.

X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB) defines a set of lower and upper bounds on the design variables X so that a local Pareto set is found in the range $LB \leq x \leq UB$. Use empty matrices for LB and UB if no bounds exist. Set LB(i) = -Inf if X(i) is unbounded below; set UB(i) = Inf if X(i) is unbounded above. Bound constraints are supported only for the default PopulationType option ('doubleVector'). Other population types, e.g., 'bitString' and 'custom', are not supported.

X = gamultiobj(FITNESSFCN,NVARS,A,b,Aeq,beq,LB,UB,options) finds a Pareto set X with the default optimization parameters replaced by values in the structure options. options can be created with the gaoptimset function.

X = gamultiobj(problem) finds the Pareto set for problem, where problem is a structure containing the following fields:

fitnessfcn	Fitness functions
nvars	Number of design variables
Aineq	A matrix for linear inequality constraints
bineq	b vector for linear inequality constraints
Aeq	A matrix for linear equality constraints
beq	b vector for linear equality constraints
lb	Lower bound on x
ub	Upper bound on x
randstate	Optional field to reset rand state

randnstate	Optional field to reset randn state
options	Options structure created using gaoptimset

Create the structure problem by exporting a problem from Optimization Tool, as described in "Importing and Exporting Your Work" in the *Optimization Toolbox™ User's Guide*.

[X,FVAL] = gamultiobj(FITNESSFCN,NVARS, ...) returns a matrix FVAL, the value of all the objective functions defined in FITNESSFCN at all the solutions in X. FVAL has numberOfObjectives columns and same number of rows as does X.

[X,FVAL,EXITFLAG] = gamultiobj(FITNESSFCN,NVARS, ...) returns EXITFLAG, which describes the exit condition of gamultiobj. Possible values of EXITFLAG and the corresponding exit conditions are listed in this table.

EXITFLAG Value	Exit Condition
1	Average change in value of the spread of Pareto set over options.StallGenLimit generations less than options.TolFun
0	Maximum number of generations exceeded
-1	Optimization terminated by the output or by the plot function
-2	No feasible point found
-5	Time limit exceeded

[X,FVAL,EXITFLAG,OUTPUT] = gamultiobj(FITNESSFCN,NVARS, ...) returns a structure OUTPUT with the following fields:

OUTPUT Field	Meaning
Output.randstate	State of the function rand used before the genetic algorithm started

OUTPUT Field	Meaning
Output.randnstate	State of the function randn used before the genetic algorithm started
Output.generations	Total number of generations, excluding HybridFcn iterations
Output.funccount	Total number of function evaluations
Output.maxconstraint	Maximum constraint violation, if any
Output.message	gamultiobj termination message

[X,FVAL,EXITFLAG,OUTPUT,POPULATION] =
gamultiobj(FITNESSFCN, ...) returns the final POPULATION at
termination.

[X,FVAL,EXITFLAG,OUTPUT,POPULATION,SCORE] =
gamultiobj(FITNESSFCN, ...) returns the SCORE of the
final POPULATION.

Example　　　**GA Multiobjective**

This example optimizes two objectives defined by Schaffer's second
function: a vector-valued function of two components and one input
argument. The Pareto front is disconnected. Define this function in
an M-file:

```
function y = schaffer2(x) % y has two columns

% Initialize y for two objectives and for all x
y = zeros(length(x),2);

% Evaluate first objective.
% This objective is piecewise continuous.
for i = 1:length(x)
    if x(i) <= 1
        y(i,1) = -x(i);
    elseif x(i) <=3
        y(i,1) = x(i) -2;
```

```
        elseif x(i) <=4
            y(i,1) = 4 - x(i);
        else
            y(i,1) = x(i) - 4;
        end
    end

    % Evaluate second objective
    y(:,2) = (x -5).^2;
```

First, plot the two objectives:

```
x = -1:0.1:8;
y = schaffer2(x);

plot(x,y(:,1),'.r'); hold on
plot(x,y(:,2),'.b');
```

The two component functions compete in the range [1, 3] and [4, 5]. But the Pareto-optimal front consists of only two disconnected regions: [1, 2] and [4, 5]. This is because the region [2, 3] is inferior to [1, 2].

Next, impose a bound constraint on x, $-5 \le x \le 10$ setting

```
lb = -5;
ub = 10;
```

The best way to view the results of the genetic algorithm is to visualize the Pareto front directly using the @gaplotpareto option. To optimize Schaffer's function, a larger population size than the default (15) is needed, because of the disconnected front. This example uses 60. Set the optimization options as:

```
options = gaoptimset('PopulationSize',60,'PlotFcns',...
@gaplotpareto);
```

Now call gamultiobj, specifying one independent variable and only the bound constraints:

```
[x,f,exitflag] = gamultiobj(@schaffer2,1,[],[],[],[],...
lb,ub,options);

Optimization terminated: average change in the spread of
Pareto solutions less than options.TolFun.

exitflag
exitflag = 1
```

The vectors x, f(:,1), and f(:,2) respectively contain the Pareto set and both objectives evaluated on the Pareto set.

Demos

The gamultiobjfitness demo solves a simple problem with one decision variable and two objectives.

The gamultiobjoptionsdemo demo shows how to set options for multiobjective optimization.

References   [1] Deb, Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

See Also   ga, gaoptimget, gaoptimset, patternsearch, @ (Special Characters), rand, randn

gaoptimget

Purpose
Obtain values of genetic algorithm options structure

Syntax
```
val = gaoptimget(options, 'name')
val = gaoptimget(options, 'name', default)
```

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.

`val = gaoptimget(options, 'name', default)` returns the `'name'` parameter, but will return the default value if the name parameter is not specified (or is `[]`) in `options`.

See Also
For more about these options, see "Genetic Algorithm Options" on page 9-23.

`ga`, `gamultiobj`, `gaoptimset`

Purpose   Create genetic algorithm options structure

Syntax   gaoptimset
                 options = gaoptimset
                 options = gaoptimset(@ga)
                 options = gaoptimset(@gamultiobj)
                 options = gaoptimset('param1',value1,'param2',value2,...)
                 options = gaoptimset(oldopts,'param1',value1,...)
                 options = gaoptimset(oldopts,newopts)

Description gaoptimset with no input or output arguments displays a complete list
                 of parameters with their valid values.

                 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 [], indicating default values will
                 be used.

                 options = gaoptimset(@ga) creates a structure called options that
                 contains the default options for the genetic algorithm.

                 options = gaoptimset(@gamultiobj) creates a structure called
                 options that contains the default options for gamultiobj.

                 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.

gaoptimset

Options   The following table lists the options you can set with `gaoptimset`. See
"Genetic Algorithm Options" on page 9-23 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}
CrossoverFcn	Handle to the function that the algorithm uses to create crossover children	@crossoverheuristic {@crossoverscattered} @crossoverintermediate @crossoversinglepoint @crossovertwopoint @crossoverarithmetic
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}
Display	Level of display	'off' | 'iter' | 'diagnose' | {'final'}
DistanceMeasureFcn	Handle to the function that computes distance measure of individuals, computed in decision variable or design space (genotype) or in function space (phenotype)	{@distancecrowding,'phenotype'}

Option	Description	Values
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	@fitscalingshiftlinear @fitscalingprop @fitscalingtop {@fitscalingrank}
Generations	Positive integer specifying the maximum number of iterations before the algorithm halts	Positive integer |{100}
HybridFcn	Handle to a function that continues the optimization after ga terminates or Cell array specifying the hybrid function and its options structure	Function handle | @fminsearch @patternsearch @fminunc @fmincon {[]} or 1-by-2 cell array | {@solver, hybridoptions}, where solver = fminsearch, patternsearch, fminunc, or fmincon {[]}
InitialPenalty	Initial value of penalty parameter	Positive scalar | {10}
InitialPopulation	Initial population used to seed the genetic algorithm; can be partial	Matrix | {[]}

Option	Description	Values
InitialScores	Initial scores used to determine fitness; can be partial	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}
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 @mutationadaptfeasible {@mutationgaussian}
OutputFcns	Functions that ga calls at each iteration	@gaoutputgen | {[]}
ParetoFraction	Scalar between 0 and 1 specifying the fraction of individuals to keep on the first Pareto front while the solver selects individuals from higher fronts	Scalar | {0.35}
PenaltyFactor	Penalty update parameter	Positive scalar | {100}

Option	Description	Values
PlotFcns	Array of handles to functions that plot data computed by the algorithm	`@gaplotbestf @gaplotbestindiv @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}
PopInitRange	Matrix or vector specifying the range of the individuals in the initial population	Matrix or vector | [0;1]
PopulationSize	Size of the population	Positive integer | {20}
PopulationType	String describing the data type of the population	`'bitstring'	'custom'	{'doubleVector'}`  Note that linear and nonlinear constraints are not satisfied when `PopulationType` is set to `'bitString'` or `'custom'`.
SelectionFcn	Handle to the function that selects parents of crossover and mutation children	`@selectionremainder @selectionuniform {@selectionstochunif} @selectionroulette @selectiontournament`

Option	Description	Values
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 | {Inf}
TimeLimit	Positive scalar. The algorithm stops after running for `TimeLimit` seconds.	Positive scalar | {Inf}
TolCon	Positive scalar. `TolCon` is used to determine the feasibility with respect to nonlinear constraints.	Positive scalar | {1e-6}
TolFun	Positive scalar. The algorithm runs until the cumulative change in the fitness function value over `StallGenLimit` is less than `TolFun`.	Positive scalar | {1e-6}
UseParallel	Compute fitness functions of a population in parallel.	'always' | {'never'}
Vectorized	String specifying whether the computation of the fitness function is vectorized	'on' | {'off'}

See Also   For more about these options, see "Genetic Algorithm Options" on page 9-23.

ga, gamultiobj, gaoptimget

patternsearch

Purpose   Find minimum of function using 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,nonlcon)
x = patternsearch(@fun,x0,A,b,Aeq,beq,LB,UB,nonlcon,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) finds the local minimum, x, to the
MATLAB® function, fun, 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. The function fun accepts a vector input and
returns a scalar function value.

Note To write a function with additional parameters to the independent
variables that can be called by patternsearch, see the section on
"Passing Extra Parameters" in the *Optimization Toolbox™ User's Guide*.

---

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 $Ax \le b$.

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`, starting at `x0`, and subject to the constraints

$$Ax \le b$$

$$Aeq * x = beq$$

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)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range $LB \ll x \ll UB$. 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,nonlcon)` subjects the minimization to the constraints defined in `nonlcon`, a nonlinear constraint function. The function `nonlcon` accepts x and returns the vectors C and Ceq, representing the nonlinear inequalities and equalities respectively. `fmincon` minimizes `fun` such that $C(x) \le 0$ and $Ceq(x) = 0$. (Set `LB = [ ]` and `UB = [ ]` if no bounds exist.)

`x = patternsearch(@fun,x0,A,b,Aeq,beq,LB,UB,nonlcon,options)` minimizes `fun` with the default optimization parameters replaced by values in `options`. The structure `options` can be created using `psoptimset`.

`x = patternsearch(problem)` finds the minimum for `problem`, where `problem` is a structure containing the following fields:

patternsearch

- `objective` — Objective function

- `X0` — Starting point

- `Aineq` — Matrix for linear inequality constraints

- `bineq` — Vector for linear inequality constraints

- `Aeq` — Matrix for linear equality constraints

- `beq` — Vector for linear equality constraints

- `lb` — Lower bound for `x`

- `ub` — Upper bound for `x`

- `nonlcon` — Nonlinear constraint function

- `Solver` — `'patternsearch'`

- `options` — Options structure created with `psoptimset`

- `randstate` — Optional field to reset the state of `rand`

- `randnstate` — Optional field to reset the state of `randn`

Create the structure `problem` by exporting a problem from the Optimization Tool, as described in "Importing and Exporting Your Work" in the *Optimization Toolbox User's Guide*.

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`. Possible values of `exitflag` and the corresponding conditions are

1	Magnitude of mesh size is less than specified tolerance and constraint violation less than `options.TolCon`.
2	Change in x less than the specified tolerance and constraint violation less than `options.TolCon`.
4	Magnitude of step smaller than machine precision and constraint violation less than `options.TolCon`.
0	Maximum number of function evaluations or iterations reached.
-1	Optimization terminated by the output or plot function.
-2	No feasible point found.

`[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 technique

- `searchmethod` — Search technique used, if any

- `iteration` — Total number of iterations

- `funccount` — Total number of function evaluations

- `meshsize` — Mesh size at x

- `maxconstraint` — Maximum constraint violation, if any

- `message` — Reason why the algorithm terminated

patternsearch

Example   Given the following constraints

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$

$$x_1 \ge 0, \ \ x_2 \ge 0,$$

the following code finds the minimum of the function, lincontest6, that is provided with your software:

```
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: mesh size less than
                              options.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*, Volume 7, Number 1, pages 1–25, 1997.

[2] Lewis, Robert Michael and Virginia Torczon, "Pattern Search Algorithms for Bound Constrained Minimization", *SIAM Journal on Optimization*, Volume 9, Number 4, pages 1082–1099, 1999.

[3] Lewis, Robert Michael and Virginia Torczon, "Pattern Search Methods for Linearly Constrained Minimization", *SIAM Journal on Optimization*, Volume 10, Number 3, pages 917–941, 2000.

[4] Audet, Charles and J. E. Dennis Jr., "Analysis of Generalized Pattern Searches", *SIAM Journal on Optimization*, Volume 13, Number 3, pages 889–903, 2003.

[5] Lewis, Robert Michael and Virginia Torczon, "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Optimization*, Volume 12, Number 4, pages 1075–1089, 2002.

[6] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *SIAM Journal on Numerical Analysis*, Volume 28, Number 2, pages 545–572, 1991.

[7] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds", *Mathematics of Computation*, Volume 66, Number 217, pages 261–288, 1997.

See Also   optimtool, psoptimget, psoptimset, ga, simulannealbnd, threshacceptbnd

psoptimget

Purpose	Obtain values of pattern search options structure
Syntax	`val = psoptimget(options,'name')` `val = psoptimget(options,'name',default)`
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. `val = psoptimget(options,'name',default)` returns the value of the parameter name from the pattern search options structure `options`, but returns default if the parameter is not specified (as in `[]`) in `options`.
Example	`val = psoptimget(opts,'TolX',1e-4);` returns `val = 1e-4` if the `TolX` property is not specified in `opts`.
See Also	For more about these options, see "Pattern Search Options" on page 9-2. `psoptimset, patternsearch`

Purpose　　Create pattern search options structure

Syntax
```
psoptimset
options = psoptimset
options = psoptimset('param1',value1,'param2',value2,...)
options = psoptimset(oldopts,'param1',value1,...)
options = psoptimset(oldopts,newopts)
```

Description　　psoptimset with no input or output arguments displays a complete list of parameters with their valid values.

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.

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 9-2 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. If the objective function is stochastic, it is advised not to use this option.	'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. Use if 'Cache' option is set to 'on'.	Positive scalar | {eps}
CompletePoll	Complete poll around current iterate	'on' | {'off'}
CompleteSearch	Complete poll around current iterate	'on' | {'off'}
Display	Level of display	'off' | 'iter' | 'diagnose' | {'final'}
InitialMeshSize	Initial mesh size for pattern algorithm	Positive scalar | {1.0}
InitialPenalty	Initial value of the penalty parameter	Positive scalar | {10}

Option	Description	Values
MaxFunEvals	Maximum number of objective function evaluations	Positive integer | {2000*numberOfVariables}
MaxIter	Maximum number of iterations	Positive integer | {100*numberOfVariables}
MaxMeshSize	Maximum mesh size used in a poll/search step	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}
MeshRotate	Rotate the pattern before declaring a point to be optimum	'off' | {'on'}
OutputFcn	Specifies a user-defined function that an optimization function calls at each iteration	@psoutputhistory | {[]}
PenaltyFactor	Penalty update parameter	Positive scalar | {100}
PlotFcn	Specifies plots of output from the pattern search	@psplotbestf | @psplotmeshsize | @psplotfuncount | @psplotbestx | {[]}
PlotInterval	Specifies that plot functions will be called at every interval	{1}

psoptimset

Option	Description	Values
PollingOrder	Order of poll directions in pattern search	'Random' | 'Success' | {'Consecutive'}
PollMethod	Polling strategy used in pattern search	{'GPSPositiveBasis2N'} | 'GPSPositiveBasisNp1'| 'MADSPositiveBasis2N' | 'MADSPositiveBasisNp1'
ScaleMesh	Automatic scaling of variables	{'on'} | 'off'
SearchMethod	Type of search used in pattern search	'GPSPositiveBasisNp1' | 'GPSPositiveBasis2N' | 'MADSPositiveBasisNp1' | 'MADSPositiveBasis2N' | @searchga | @searchlhs | @searchneldermead| {[]}
TimeLimit	Total time (in seconds) allowed for optimization	Positive scalar | {Inf}
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}
UseParallel	Compute objective functions of a poll or search in parallel.	'always' | {'never'}
Vectorized	Specifies whether functions are vectorized	'on' | {'off'}

See Also   patternsearch, psoptimget

Purpose   Values of simulated annealing or threshold acceptance algorithm options structure

Syntax
```
val = saoptimget(options, 'name')
val = saoptimget(options, 'name', default)
```

Description   `val = saoptimget(options, 'name')` returns the value of the parameter `name` from the simulated annealing or threshold acceptance algorithm options structure `options`. `saoptimget(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 the parameter. `saoptimget` ignores case in parameter names.

`val = saoptimget(options, 'name', default)` returns the `'name'` parameter, but returns the default value if the `'name'` parameter is not specified (or is `[]`) in `options`.

Example
```
opts = saoptimset('TolFun',1e-4);
val = saoptimget(opts,'TolFun');
```

returns `val = 1e-4` for `TolFun`.

See Also   For more about these options, see "Simulated Annealing and Threshold Acceptance Algorithm Options" on page 9-46.

`saoptimset`, `simulannealbnd`, `threshacceptbnd`

saoptimset

Purpose   Create simulated annealing algorithm or threshold acceptance options structure

Syntax
```
saoptimset
options = saoptimset
options = saoptimset('param1',value1,'param2',value2,...)
options = saoptimset(oldopts,'param1',value1,...)
options = saoptimset(oldopts,newopts)
options = saoptimset(optimfunction)
```

Description   saoptimset with no input or output arguments displays a complete list of parameters with their valid values.

options = saoptimset (with no input arguments) creates a structure called options that contains the options, or *parameters*, for the simulated annealing or threshold acceptance algorithm with all parameters set to [].

options = saoptimset('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 []. It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names. Note that for string values, correct case and the complete string are required.

options = saoptimset(oldopts,'param1',value1,...) creates a copy of oldopts, modifying the specified parameters with the specified values.

options = saoptimset(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 = saoptimset(optimfunction) creates an options structure with all the parameter names and default values relevant to the optimization function optimfunction. optimfunction can be either 'simulannealbnd' or 'threshacceptbnd'. For example,

```
saoptimset('threshacceptbnd')

ans =
            AnnealingFcn: @annealingfast
          TemperatureFcn: @temperatureexp
           AcceptanceFcn: @acceptancethresh
                  TolFun: 1.0000e-006
           StallIterLimit: '500*numberofvariables'
              MaxFunEvals: '3000*numberofvariables'
                TimeLimit: Inf
                  MaxIter: Inf
           ObjectiveLimit: -Inf
                  Display: 'final'
          DisplayInterval: 10
                HybridFcn: []
           HybridInterval: 'end'
                  PlotFcns: []
              PlotInterval: 1
                OutputFcns: []
       InitialTemperature: 100
          ReannealInterval: 100
                  DataType: 'double'
```

Options   The following table lists the options you can set with saoptimset. See
Chapter 9, "Options Reference" for a complete description of these
options and their values. Values in {} denote the default value. You
can also view the options parameters by typing saoptimset at the
command line.

Option	Description	Values
AcceptanceFcn	Handle to the function the algorithm uses to determine if a new point is accepted	Function handle | @acceptancethresh |{@acceptancesa} for simulannealbnd or {@acceptancethresh} for threshacceptbnd

Option	Description	Values
AnnealingFcn	Handle to the function the algorithm uses to generate new points	Function handle | @annealingboltz | {@annealingfast}
DataType	Type of decision variable	'custom' | {'double'}
Display	Level of display	'off' | 'iter' | 'diagnose' | {'final'}
DisplayInterval	Interval for iterative display	Positive integer | {10}
HybridFcn	Automatically run HybridFcn (another optimization function) during or at the end of iterations of the solver	Function handle | @fminsearch @patternsearch @fminunc @fmincon {[]} or 1-by-2 cell array | {@solver, hybridoptions}, where solver = fminsearch, patternsearch, fminunc, or fmincon {[]}
HybridInterval	Interval (if not 'end' or 'never') at which HybridFcn is called	Positive integer | 'never' | {'end'}
InitialTemperature	Initial value of temperature	Positive integer | {100}
MaxFunEvals	Maximum number of objective function evaluations allowed	Positive scalar | {3000*numberOfVariables}
MaxIter	Maximum number of iterations allowed	Positive scalar | {Inf}
ObjectiveLimit	Minimum objective function value desired	Scalar | {Inf}
OutputFcns	Function(s) get(s) iterative data and can change options at run time	Function handle or cell array of function handles | {[]}

Option	Description	Values
PlotFcns	Plot function(s) called during iterations	Function handle or cell array of function handles | @saplotbestf | @saplotbestx | @saplotf | @saplotstopping | @saplottemperature | {[]}
PlotInterval	Plot functions are called at every interval	Positive integer | {1}
ReannealInterval	Reannealing interval	Positive integer | {100}
StallIterLimit	Number of iterations over which average change in fitness function value at current point is less than options.TolFun	Positive integer | {500*numberOfVariables}
TemperatureFcn	Function used to update temperature schedule	Function handle | @temperatureboltz | @temperaturefast | {@temperatureexp}
TimeLimit	The algorithm stops after running for TimeLimit seconds	Positive scalar | {Inf}
TolFun	Termination tolerance on function value	Positive scalar | {1e-6}

See Also   For more about these options, see "Simulated Annealing and Threshold Acceptance Algorithm Options" on page 9-46.

saoptimget, simulannealbnd, threshacceptbnd

simulannealbnd

Purpose	Find unconstrained or bound-constrained minimum of function of several variables using simulated annealing algorithm

Syntax	`x = simulannealbnd(fun,x0)`
	`x = simulannealbnd(fun,x0,lb,ub)`
	`x = simulannealbnd(fun,x0,lb,ub,options)`
	`x = simulannealbnd(problem)`
	`[x,fval] = simulannealbnd(...)`
	`[x,fval,exitflag] = simulannealbnd(...)`
	`[x,fval,exitflag,output] = simulannealbnd(fun,...)`

Description   `x = simulannealbnd(fun,x0)` starts at `x0` and finds a local minimum `x` to the objective function specified by the function handle `fun`. The objective function accepts input `x` and returns a scalar function value evaluated at `x`. `x0` may be a scalar or a vector.

`x = simulannealbnd(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that a solution is found in the range `lb` ≤ `x` ≤ `ub`. Use empty matrices for `lb` and `ub` if no bounds exist. Set `lb(i)` to `-Inf` if `x(i)` is unbounded below; set `ub(i)` to `Inf` if `x(i)` is unbounded above.

`x = simulannealbnd(fun,x0,lb,ub,options)` minimizes with the default optimization parameters replaced by values in the structure `options`, which can be created using the `saoptimset` function. See the `saoptimset` reference page for details.

`x = simulannealbnd(problem)` finds the minimum for `problem`, where `problem` is a structure containing the following fields:

`objective`	Objective function
`x0`	Initial point of the search
`lb`	Lower bound on `x`
`ub`	Upper bound on `x`
`randstate`	Optional field to reset `rand` state

randnstate	Optional field to reset randn state
solver	'simulannealbnd'
options	Options structure created using saoptimset

Create the structure problem by exporting a problem from Optimization Tool, as described in "Importing and Exporting Your Work" in the *Optimization Toolbox™ User's Guide*.

[x,fval] = simulannealbnd(...) returns fval, the value of the objective function at x.

[x,fval,exitflag] = simulannealbnd(...) returns exitflag, an integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated:

- 1 — Average change in the value of the objective function over options.StallIterLimit iterations is less than options.TolFun.

- 5 — options.ObjectiveLimit limit reached.

- 0 — Maximum number of function evaluations or iterations exceeded.

- -1 — Optimization terminated by the output or plot function.

- -2 — No feasible point found.

- -5 — Time limit exceeded.

[x,fval,exitflag,output] = simulannealbnd(fun,...) returns output, a structure that contains information about the problem and the performance of the algorithm. The output structure contains the following fields:

- problemtype — Type of problem: unconstrained or bound constrained.

- iterations — The number of iterations computed.

- funccount — The number of evaluations of the objective function.

- message — The reason the algorithm terminated.

- temperature — Temperature when the solver terminated.

- totaltime — Total time for the solver to run.

- 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 simulannealbnd. See "Reproducing Your Results" on page 7-5.

Examples   Minimization of De Jong's fifth function, a two-dimensional function with many local minima. Enter the command dejong5fcn to generate the following plot.

```
x0 = [0 0];
[x,fval] = simulannealbnd(@dejong5fcn,x0)

Optimization terminated: change in best function value
                         less than options.TolFun.

x =
    0.0392  -31.9700

fval =
    2.9821
```

Minimization of De Jong's fifth function subject to lower and upper bounds:

```
x0 = [0 0];
lb = [-64 -64];
ub = [64 64];
[x,fval] = simulannealbnd(@dejong5fcn,x0,lb,ub)

Optimization terminated: change in best function value
                          less than options.TolFun.

x =
  -31.9652  -32.0286

fval =
    0.9980
```

The objective can also be an anonymous function:

```
fun =  @(x) 3*sin(x(1))+exp(x(2));
x = simulannealbnd(fun,[1;1],[0 0])

Optimization terminated: change in best function value
                          less than options.TolFun.

x =
  457.1045
    0.0000
```

Minimization of De Jong's fifth function while displaying plots:

```
x0 = [0 0];
options = saoptimset('PlotFcns',{@saplotbestx,...
                @saplotbestf,@saplotx,@saplotf});
simulannealbnd(@dejong5fcn,x0,[],[],options)

Optimization terminated: change in best function value
                          less than options.TolFun.
```

```
ans =
     0.0230  -31.9806
```

The plots displayed are shown below.

ga, patternsearch, saoptimget, saoptimset, threshacceptbnd

Purpose   Find unconstrained or bound-constrained minimum of function of several variables using threshold acceptance algorithm

Syntax
```
x = threshacceptbnd(fun,x0)
x = threshacceptbnd(fun,x0,lb,ub)
x = threshacceptbnd(fun,x0,lb,ub,options)
x = threshacceptbnd(problem)
[x,fval] = threshacceptbnd(...)
[x,fval,exitflag] = threshacceptbnd(...)
[x,fval,exitflag,output] = threshacceptbnd(...)
```

Description   x = threshacceptbnd(fun,x0) starts at x0 and finds a local minimum x to the objective function specified by the function handle fun. The objective function accepts input x and returns a scalar function value evaluated at x. x0 may be a scalar or a vector.

x = threshacceptbnd(fun,x0,lb,ub) defines a set of lower and upper bounds on the design variables, x, so that a solution is found in the range lb ≤ x ≤ ub. Use empty matrices for lb and ub if no bounds exist. Set lb(i) to -Inf if x(i) is unbounded below; set ub(i) to Inf if x(i) is unbounded above.

x = threshacceptbnd(fun,x0,lb,ub,options) minimizes with the default optimization parameters replaced by values in the structure options, which can be created using the saoptimset function. See the saoptimset reference page for details.

x = threshacceptbnd(problem) finds the minimum for problem, where problem is a structure containing the following fields:

objective	Objective function
x0	Initial point of the search
lb	Lower bound on x
ub	Upper bound on x
randstate	Optional field to reset rand state

randnstate	Optional field to reset `randn` state
solver	`'threshacceptbnd'`
options	Options structure created using `saoptimset`

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Importing and Exporting Your Work" in the *Optimization Toolbox™ User's Guide*.

`[x,fval] = threshacceptbnd(...)` returns `fval`, the value of the objective function at `x`.

`[x,fval,exitflag] = threshacceptbnd(...)` returns `exitflag`, an integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated:

- `1` — Average change in the value of the objective function over `options.StallIterLimit` iterations is less than `options.TolFun`.

- `5` — `options.ObjectiveLimit` limit reached.

- `0` — Maximum number of function evaluations or iterations exceeded.

- `-1` — Optimization terminated by the output or plot function.

- `-2` — No feasible point found.

- `-5` — Time limit exceeded.

`[x,fval,exitflag,output] = threshacceptbnd(...)` returns `output`, a structure that contains information about the problem and the performance of the algorithm. The `output` structure contains the following fields:

- `problemtype` — Type of problem: unconstrained or bound constrained.

- `iterations` — The number of iterations computed.

- `funccount` — The number of evaluations of the objective function.

- message — The reason the algorithm terminated.

- temperature — Temperature when the solver terminated.

- totaltime — Total time for the solver to run.

- 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 simulannealbnd. See "Reproducing Your Results" on page 7-5.

Examples   Minimization of De Jong's fifth function, a two-dimensional function with many local minima. Enter the command dejong5fcn to generate the following plot.

```
x0 = [0 0];
[x,fval] = simulannealbnd(@dejong5fcn,x0)

Optimization terminated: change in best function value
                         less than options.TolFun.

x =
    0.0392  -31.9700

fval =
    2.9821
```

Minimization of De Jong's fifth function subject to lower and upper bounds:

```
x0 = [0 0];
lb = [-64 -64];
ub = [64 64];
[x,fval] = simulannealbnd(@dejong5fcn,x0,lb,ub)

Optimization terminated: change in best function value
                        less than options.TolFun.

x =
  -31.9652  -32.0286

fval =
    0.9980
```

The objective can also be an anonymous function:

```
fun =  @(x) 3*sin(x(1))+exp(x(2));
x = simulannealbnd(fun,[1;1],[0 0])

Optimization terminated: change in best function value
                        less than options.TolFun.

x =
  457.1045
    0.0000
```

Minimization of De Jong's fifth function while displaying plots:

```
x0 = [0 0];
options = saoptimset('PlotFcns',{@saplotbestx,...
                @saplotbestf,@saplotx,@saplotf});
simulannealbnd(@dejong5fcn,x0,[],[],options)

Optimization terminated: change in best function value
                        less than options.TolFun.
```

```
ans =
    0.0230  -31.9806
```

The plots displayed are shown below.

See Also   saoptimget, saoptimset, threshacceptbnd, patternsearch, ga

Examples

Use this list to find examples in the documentation.

Pattern Search

Genetic Algorithm

Simulated Annealing and Threshold Acceptance Algorithms

Index

A

accelerator
  mesh 5-31
algorithm
  genetic 3-20
  pattern search 2-15
  simulated annealing 4-13
  threshold acceptance 4-13
annealing 4-11
annealing schedule 4-11

C

cache 5-32
children
  crossover 3-22
  elite 3-22
  in genetic algorithms 3-19
  mutation 3-22
creation function 6-23 6-26
Creation function 9-27
crossover 6-35
  children 3-22
  fraction 6-38

D

direct search 2-2
diversity 3-18

E

elite children 3-22
example
  genetic algorithm 3-8 6-3 6-7
  GPS algorithm 2-7 5-2
  simulated annealing algorithm 4-8 7-8
  threshold acceptance algorithm 7-8
expansion
  mesh 5-26

F

fitness function 3-17
  vectorizing 5-42 6-54
  writing M-files 1-3
fitness scaling 6-31

G

ga function 11-2
gamultiobj function 11-8
gaoptimget function 11-14
gaoptimset function 11-15
generations 3-18
genetic algorithm
  description 3-20
  nonlinear constraint algorithm, ALGA 3-27
  options 9-23
  overview 3-2
  setting options at command line 6-13
  stopping criteria 3-24
  using from command line 6-12
global and local minima 6-44
GPS 2-2

H

hybrid function 6-49

I

individuals
  applying the fitness function 3-17
initial population 3-21

M

M-files
  writing 1-3
MADS 2-2
maximizing functions 1-4
mesh 2-13