Global Optimization Toolbox 3
User's Guide

MATLAB®

The MathWorks™

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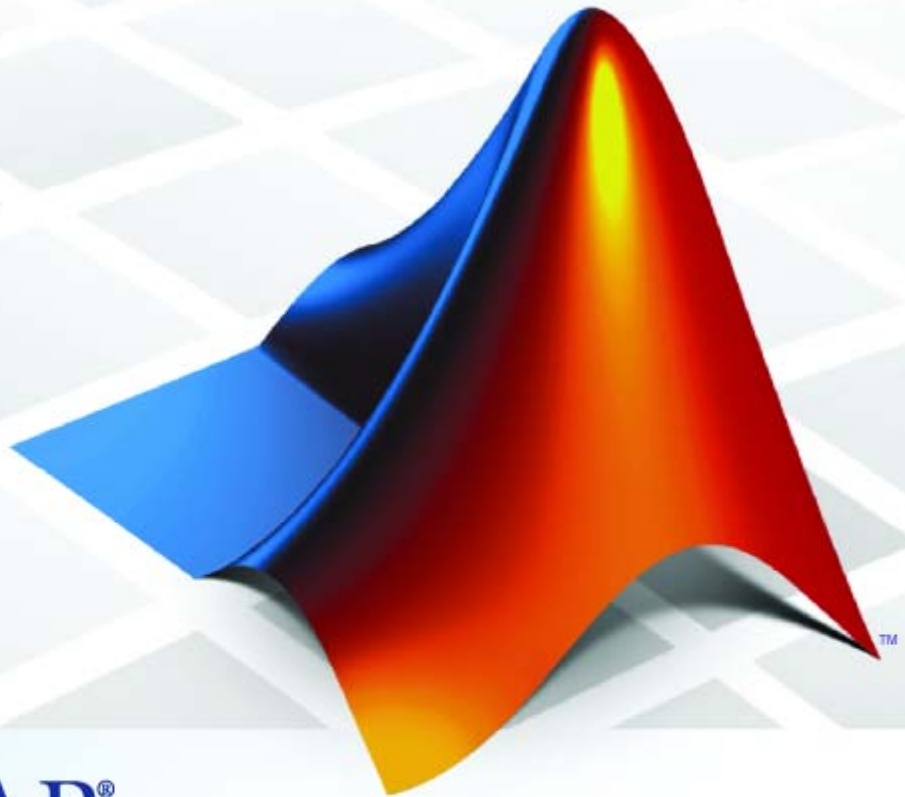
How to Contact The MathWorks

	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.

Global Optimization Toolbox User's Guide

© COPYRIGHT 2004–2010 by The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Contents

Using GlobalSearch and MultiStart

3

Using Direct Search

4

Using the Genetic Algorithm

5

Using Simulated Annealing

6

Multiobjective Optimization

7

Parallel Processing

8

Options Reference

9

Function Reference

10

Class Reference

11

Functions — Alphabetical List

12

Examples

A

Index
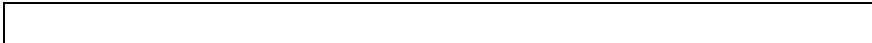
Introducing Global Optimization Toolbox Functions

Product Overview

Global Optimization Toolbox provides methods that search for global solutions to problems that contain multiple maxima or minima. It includes global search, multistart, pattern search, genetic algorithm, and simulated annealing solvers. You can use these solvers to solve optimization problems where the objective or constraint function is continuous, discontinuous, stochastic, does not possess derivatives, or includes simulations or black-box functions with undefined values for some parameter settings.

Genetic algorithm and pattern search solvers support algorithmic customization. You can create a custom genetic algorithm variant by modifying initial population and fitness scaling options or by defining parent selection, crossover, and mutation functions. You can customize pattern search by defining polling, searching, and other functions.

Key features:

- Interactive tools for defining and solving optimization problems and monitoring solution progress

- Global search and multistart solvers for finding single or multiple global optima

- Genetic algorithm solver that supports linear, nonlinear, and bound constraints

- Multiobjective genetic algorithm with Pareto-front identification, including linear and bound constraints

- Pattern search solver that supports linear, nonlinear, and bound constraints

- Simulated annealing tools that implement a random search method, with options for defining annealing process, temperature schedule, and acceptance criteria

- Parallel computing support in multistart, genetic algorithm, and pattern search solver

- Custom data type support in genetic algorithm, multiobjective genetic algorithm, and simulated annealing solvers

Implementation Notes

Global Optimization Toolbox solvers repeatedly attempt to locate a global solution. However, no solver employs an algorithm that can certify a solution as global.

You can extend the capabilities of Global Optimization Toolbox functions by writing your own files, by using them in combination with other toolboxes, or with the MATLAB® or Simulink® environments.

Example: Comparing Several Solvers

Function to Optimize

This example shows how to minimize Rastrigin's function with four solvers. Each solver has its own characteristics. The characteristics lead to different solutions and run times. The results, examined in "Comparison of Syntax and Solutions" on page 1-10, can help you choose an appropriate solver for your own problems.

Rastrigin's function has many local minima, with a global minimum at (0,0):

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

Usually you don't know the location of the global minimum of your objective function. To show how the solvers look for a global solution, this example starts all the solvers around the point [20,30], which is far from the global minimum.

The `rastriginsfcn.m` file implements Rastrigin's function. This file comes with Global Optimization Toolbox software. This example employs a scaled version of Rastrigin's function with larger basins of attraction. For information, see "Basins of Attraction" on page 1-13.

```
rf2 = @(x)rastriginsfcn(x/10);
```

rastriginsfcn([x/10,y/10])
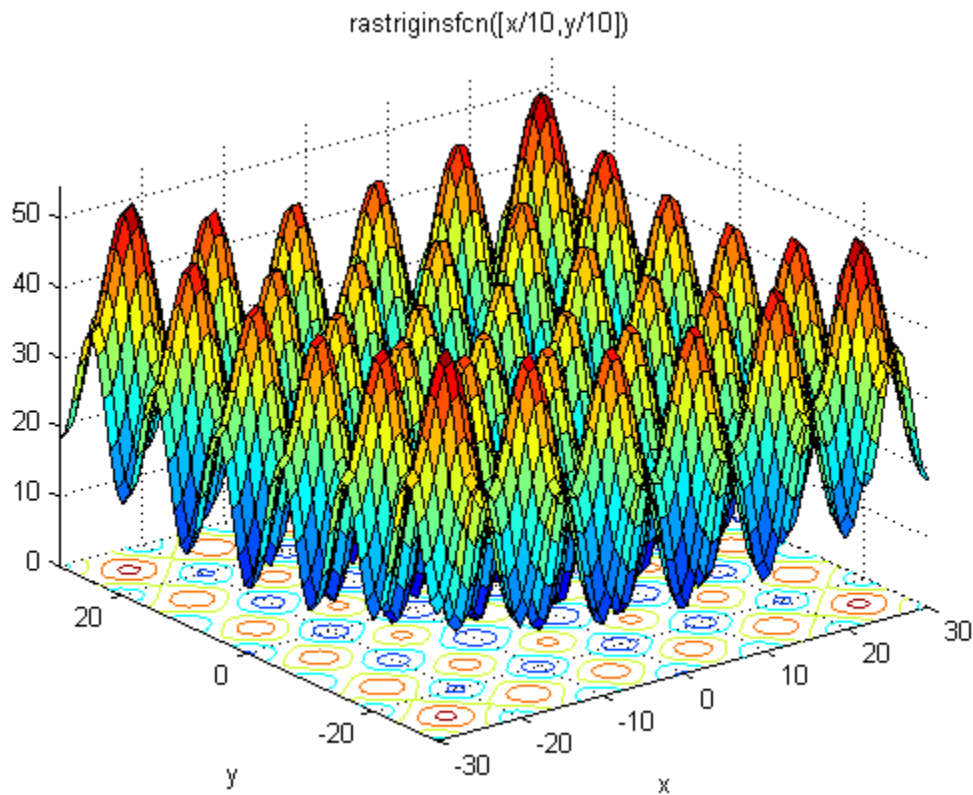
This example minimizes rf2 using the default settings of fminunc (an Optimization Toolbox™ solver), patternsearch, and GlobalSearch. It also uses ga with a nondefault setting, to obtain an initial population around the point [20,30].

Four Solution Methods

- "fminunc" on page 1-6
- "patternsearch" on page 1-7
- "ga" on page 1-8

- "GlobalSearch" on page 1-9

fminunc

To solve the optimization problem using the fminunc Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
[xf ff flf of] = fminunc(rf2,x0)
```

fminunc returns

```
Warning: Gradient must be provided for trust-region algorithm;
  using line-search algorithm instead.
> In fminunc at 347

Local minimum found.

Optimization completed because the size of the gradient is
less than the default value of the function tolerance.

xf =
   19.8991   29.8486
ff =
   12.9344
flf =
    1
of =
        iterations: 3
         funcCount: 15
          stepsize: 1
     firstorderopt: 5.9907e-009
         algorithm: 'medium-scale: Quasi-Newton line search'
           message: [1x438 char]
```

- xf is the minimizing point.

- ff is the value of the objective, rf2, at xf.

- flf is the exit flag. An exit flag of 1 indicates xf is a local minimum.

- `of` is the output structure, which describes the `fminunc` calculations leading to the solution.

patternsearch

To solve the optimization problem using the `patternsearch` Global Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
[xp fp flp op] = patternsearch(rf2,x0)
```

`patternsearch` returns

```
Optimization terminated: mesh size less than options.TolMesh.
xp =
   19.8991   -9.9496
fp =
    4.9748
flp =
     1
op =
         function: @(x)rastriginsfcn(x/10)
      problemtype: 'unconstrained'
       pollmethod: 'gpspositivebasis2n'
     searchmethod: []
       iterations: 48
        funccount: 174
         meshsize: 9.5367e-007
          message: 'Optimization terminated: mesh size less than
                    options.TolMesh.'
```

- `xp` is the minimizing point.

- `fp` is the value of the objective, `rf2`, at `xp`.

- `flp` is the exit flag. An exit flag of 1 indicates `xp` is a local minimum.

- `op` is the output structure, which describes the `patternsearch` calculations leading to the solution.

ga

To solve the optimization problem using the ga Global Optimization Toolbox solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
initpop = 10*randn(20,2) + repmat([10 30],20,1);
opts = gaoptimset('InitialPopulation',initpop);
[xga fga flga oga] = ga(rf2,2,[],[],[],[],[],[],[],opts)
```

initpop is a 20-by-2 matrix. Each row of initpop has mean [10,30], and each element is normally distributed with standard deviation 10. The rows of initpop form an initial population matrix for the ga solver.

opts is an optimization structure setting initpop as the initial population.

The final line calls ga, using the optimization structure.

ga uses random numbers, and produces a random result. In this case ga returns:

```
Optimization terminated: average change in the fitness value
less than options.TolFun.
xga =
   -0.0016   19.8821
fga =
    3.9804
flga =
     1
oga =
    problemtype: 'unconstrained'
      rngstate: [1x1 struct]
    generations: 54
      funccount: 1100
        message: [1x86 char]
```

- xga is the minimizing point.

- fga is the value of the objective, rf2, at xga.

- flga is the exit flag. An exit flag of 1 indicates xga is a local minimum.

- oga is the output structure, which describes the ga calculations leading to the solution.

GlobalSearch

To solve the optimization problem using the GlobalSearch solver, enter:

```
rf2 = @(x)rastriginsfcn(x/10); % objective
x0 = [20,30]; % start point away from the minimum
problem = createOptimProblem('fmincon','objective',rf2,...
    'x0',x0);
gs = GlobalSearch;
[xg fg flg og] = run(gs,problem)
```

problem is an optimization problem structure. problem specifies the fmincon solver, the rf2 objective function, and x0 = [20,30]. For more information on using createOptimProblem, see "Create a Problem Structure" on page 3-4.

Note You must specify fmincon as the solver for GlobalSearch, even for unconstrained problems.

gs is a default GlobalSearch object. The object contains options for solving the problem. Calling run(gs,problem) runs problem from multiple start points. The start points are random, so the following result is also random.

In this case, the run returns:

```
GlobalSearch stopped because it analyzed all the trial points.

All 6 local solver runs converged with a positive local solver exit flag.

xg =
  1.0e-007 *
   -0.1405   -0.1405
fg =
     0
flg =
     1
og =
```

```
                        funcCount: 2312
                  localSolverTotal: 6
                localSolverSuccess: 6
             localSolverIncomplete: 0
             localSolverNoSolution: 0
                           message: [1x137 char]
```

- `xg` is the minimizing point.

- `fg` is the value of the objective, `rf2`, at `xg`.

- `flg` is the exit flag. An exit flag of `1` indicates all `fmincon` runs converged properly.

- `og` is the output structure, which describes the `GlobalSearch` calculations leading to the solution.

Comparison of Syntax and Solutions

One solution is better than another if its objective function value is smaller than the other. The following table summarizes the results, accurate to one decimal.

Results	fminunc	patternsearch	ga	GlobalSearch
solution	[19.9 29.9]	[19.9 -9.9]	[0 19.9]	[0 0]
objective	12.9	5	4	0
# Fevals	15	174	1040	2312

These results are typical:

- `fminunc` quickly reaches the local solution within its starting basin, but does not explore outside this basin at all. `fminunc` has simple calling syntax.

- `patternsearch` takes more function evaluations than `fminunc`, and searches through several basins, arriving at a better solution than `fminunc`. `patternsearch` calling syntax is the same as that of `fminunc`.

- `ga` takes many more function evaluations than `patternsearch`. By chance it arrived at a slightly better solution. `ga` is stochastic, so its results change

with every run. `ga` has simple calling syntax, but there are extra steps to have an initial population near `[20,30]`.

- `GlobalSearch` `run` takes many more function evaluations than `patternsearch`, searches many basins, and arrives at an even better solution. In this case, `GlobalSearch` found the global optimum. Setting up `GlobalSearch` is more involved than setting up the other solvers. As the example shows, before calling `GlobalSearch`, you must create both a `GlobalSearch` object (`gs` in the example), and a problem structure (`problem`). Then, call the `run` method with `gs` and `problem`. For more details on how to run `GlobalSearch`, see "How to Optimize with GlobalSearch and MultiStart" on page 3-2.

What Is Global Optimization?

In this section...
"Local vs. Global Optima" on page 1-12
"Basins of Attraction" on page 1-13

Local vs. Global Optima

Optimization is the process of finding the point that minimizes a function. More specifically:

- A *local* minimum of a function is a point where the function value is smaller than or equal to the value at nearby points, but possibly greater than at a distant point.

- A *global* minimum is a point where the function value is smaller than or equal to the value at all other feasible points.
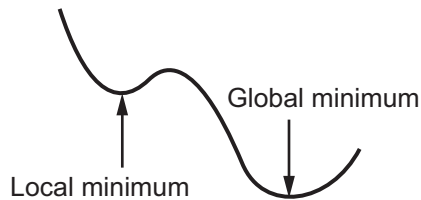
Generally, Optimization Toolbox solvers find a local optimum. (This local optimum can be a global optimum.) They find the optimum in the *basin of attraction* of the starting point. For more information, see "Basins of Attraction" on page 1-13.

In contrast, Global Optimization Toolbox solvers are designed to search through more than one basin of attraction. They search in various ways:

- GlobalSearch and MultiStart generate a number of starting points. They then use a local solver to find the optima in the basins of attraction of the starting points.

- `ga` uses a set of starting points (called the population) and iteratively generates better points from the population. As long as the initial population covers several basins, `ga` can examine several basins.

- `simulannealbnd` performs a random search. Generally, `simulannealbnd` accepts a point if it is better than the previous point. `simulannealbnd` occasionally accepts a worse point, in order to reach a different basin.

- `patternsearch` looks at a number of neighboring points before accepting one of them. If some neighboring points belong to different basins, `patternsearch` in essence looks in a number of basins at once.

Basins of Attraction

If an objective function $f(x)$ is smooth, the vector $-\nabla f(x)$ points in the direction where $f(x)$ decreases most quickly. The equation of steepest descent, namely

$$\frac{d}{dt}x(t) = -\nabla f(x(t)),$$

yields a path $x(t)$ that goes to a local minimum as t gets large. Generally, initial values $x(0)$ that are close to each other give steepest descent paths that tend to the same minimum point. The *basin of attraction* for steepest descent is the set of initial values leading to the same local minimum.

The following figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and it shows directions of steepest descent with arrows. For this and subsequent figures, black dots represent local minima. Every steepest descent path, starting at a point $x(0)$, goes to the black dot in the basin containing $x(0)$.
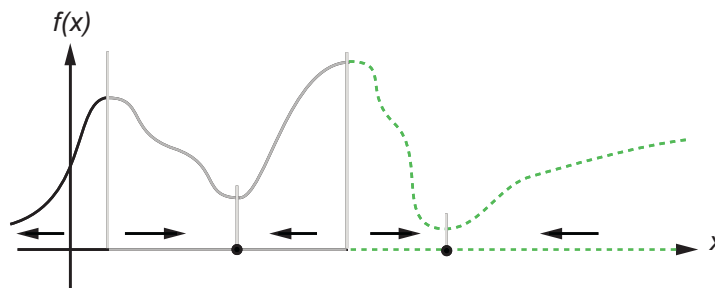
The following figure shows how steepest descent paths can be more complicated in more dimensions.
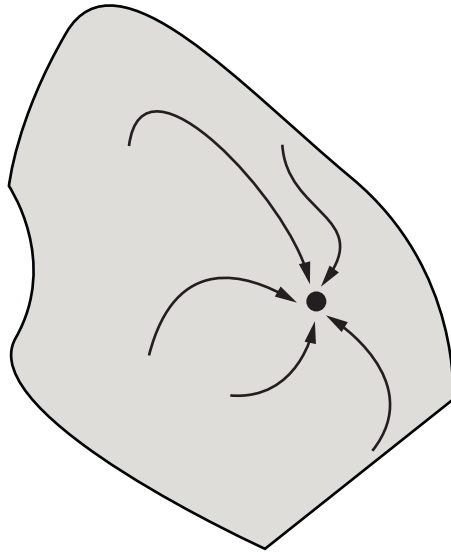
The following figure shows even more complicated paths and basins of attraction.
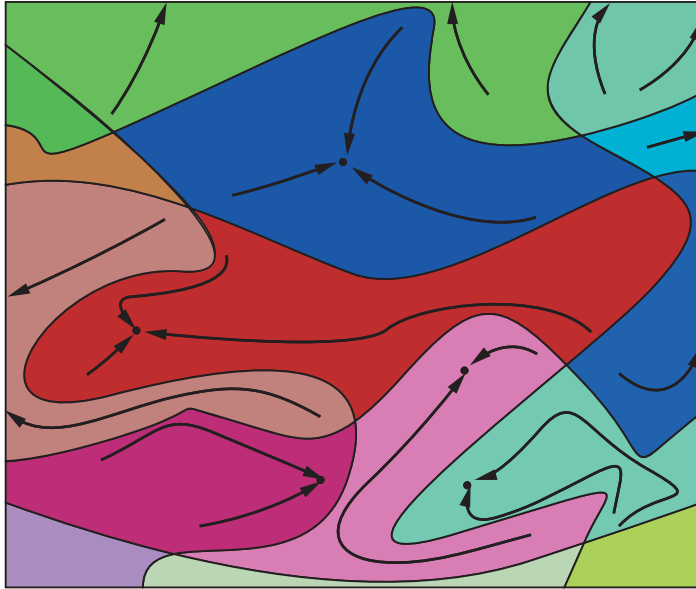
Constraints can break up one basin of attraction into several pieces. For example, consider minimizing y subject to:

- $y \geq |x|$

- $y \geq 5 - 4(x-2)^2$.

The figure shows the two basins of attraction with the final points.
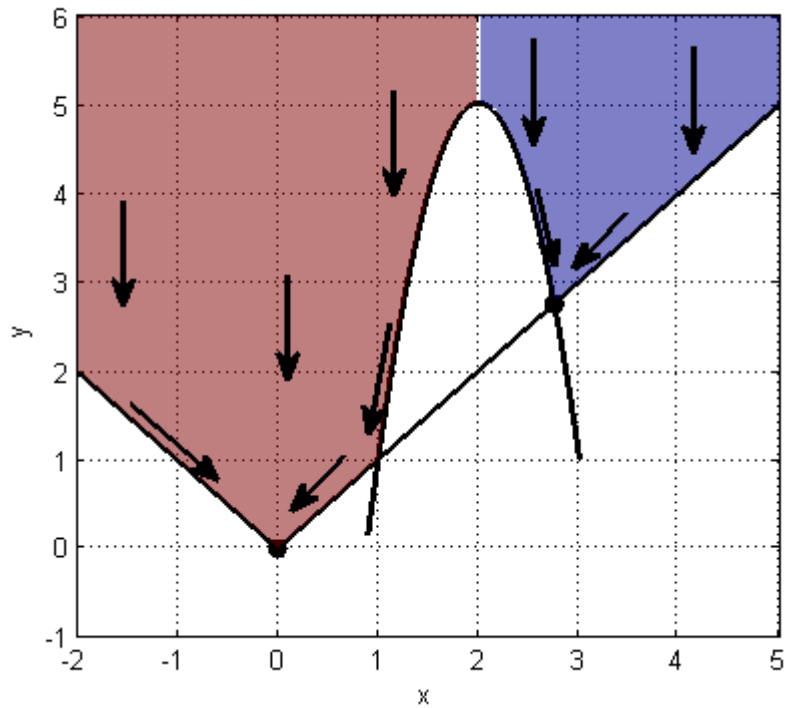
The steepest descent paths are straight lines down to the constraint boundaries. From the constraint boundaries, the steepest descent paths travel down along the boundaries. The final point is either (0,0) or (11/4,11/4), depending on whether the initial *x*-value is above or below 2.

Choosing a Solver

In this section...
"Table for Choosing a Solver" on page 1-17
"Solver Characteristics" on page 1-22
"Why Are Some Solvers Objects?" on page 1-24

Table for Choosing a Solver

There are six Global Optimization Toolbox solvers:

- `ga` (Genetic Algorithm)
- `GlobalSearch`
- `MultiStart`
- `patternsearch`, also called direct search
- `simulannealbnd` (Simulated Annealing)
- `gamultiobj`, which is not a minimizer; see Chapter 7, "Multiobjective Optimization"

Choose an optimizer based on problem characteristics and on the type of solution you want. "Solver Characteristics" on page 1-22 contains more information that can help you decide which solver is likely to be most suitable.

- "Explanation of "Desired Solution"" on page 1-18
- "Choosing Between Solvers for Smooth Problems" on page 1-20
- "Choosing Between Solvers for Nonsmooth Problems" on page 1-21

Desired Solution	Smooth Objective and Constraints	Nonsmooth Objective or Constraints
Single local solution	Optimization Toolbox functions; see "Optimization Decision Table" in the Optimization Toolbox documentation	`fminbnd`, `patternsearch`, `fminsearch`, `ga`, `simulannealbnd`
Multiple local solutions	`GlobalSearch`, `MultiStart`	
Single global solution	`GlobalSearch`, `MultiStart`, `patternsearch`, `ga`, `simulannealbnd`	`patternsearch`, `ga`, `simulannealbnd`
Single local solution using parallel processing	`MultiStart`, Optimization Toolbox functions	`patternsearch`, `ga`
Multiple local solutions using parallel processing	`MultiStart`	
Single global solution using parallel processing	`MultiStart`	`patternsearch`, `ga`

Explanation of "Desired Solution"

To understand the meaning of the terms in "Desired Solution," consider the example

$$f(x) = 100x^2(1-x)^2 - x,$$

which has local minima x1 near 0 and x2 near 1:

$$100 \, x^2 \, (1-x)^2 - x$$
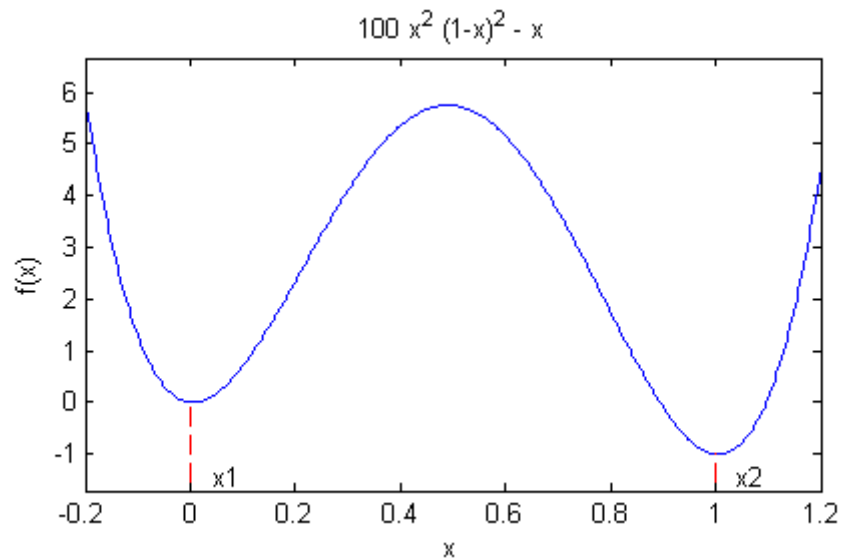
The minima are located at:

```
x1 = fminsearch(@(x)(100*x^2*(x - 1)^2 - x),0)
x1 =
    0.0051

x2 = fminsearch(@(x)(100*x^2*(x - 1)^2 - x),1)
x2 =
    1.0049
```

Description of the Terms

Term	Meaning
Single local solution	Find one local solution, a point x where the objective function $f(x)$ is a local minimum. For more details, see "Local vs. Global Optima" on page 1-12. In the example, both x1 and x2 are local solutions.
Multiple local solutions	Find a set of local solutions. In the example, the complete set of local solutions is {x1,x2}.

Description of the Terms (Continued)

Term	Meaning
Single global solution	Find the point x where the objective function $f(x)$ is a global minimum. In the example, the global solution is x2.

Choosing Between Solvers for Smooth Problems

Single Global Solution.

1 Try GlobalSearch first. It is most focused on finding a global solution, and has an efficient local solver, fmincon.

2 Try MultiStart second. It has efficient local solvers, and can search a wide variety of start points.

3 Try patternsearch third. It is less efficient, since it does not use gradients. However, patternsearch is robust and is more efficient than the remaining local solvers.

4 Try ga fourth. It can handle all types of constraints, and is usually more efficient than simulannealbnd.

5 Try simulannealbnd last. It can handle only unconstrained or bound constraints. simulannealbnd is usually the least efficient solver. However, given a slow enough cooling schedule, it can find a global solution.

Multiple Local Solutions. GlobalSearch and MultiStart both provide multiple local solutions. For the syntax to obtain multiple solutions, see "Multiple Solutions" on page 3-24. GlobalSearch and MultiStart differ in the following characteristics:

- MultiStart can find more local minima. This is because GlobalSearch rejects many generated start points (initial points for local solution). Essentially, GlobalSearch accepts a start point only when it determines that the point has a good chance of obtaining a global minimum. In contrast, MultiStart passes all generated start points to a local solver. For more information, see "GlobalSearch Algorithm" on page 3-37.

- `MultiStart` offers a choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`. The `GlobalSearch` solver uses only `fmincon` as its local solver.

- `GlobalSearch` uses a scatter-search algorithm for generating start points. In contrast, `MultiStart` generates points uniformly at random within bounds, or allows you to provide your own points.

- `MultiStart` can run in parallel. See "How to Use Parallel Processing" on page 8-11.

Choosing Between Solvers for Nonsmooth Problems

Choose the applicable solver with the lowest number:

1 Use `fminbnd` first on one-dimensional bounded problems only. `fminbnd` provably converges quickly in one dimension.

2 Use `patternsearch` on any other type of problem. `patternsearch` provably converges, and handles all types of constraints.

3 Try `fminsearch` next for low-dimensional unbounded problems. `fminsearch` is not as general as `patternsearch` and can fail to converge. For low-dimensional problems, `fminsearch` is simple to use, since it has few tuning options.

4 Try `ga` next. `ga` has little supporting theory and is often less efficient than `patternsearch`. It handles all types of constraints.

5 Try `simulannealbnd` last for unbounded problems, or for problems with bounds. `simulannealbnd` provably converges only for a logarithmic cooling schedule, which is extremely slow. `simulannealbnd` takes only bound constraints, and is often less efficient than `ga`.

Solver Characteristics

Solver	Convergence	Characteristics
`GlobalSearch`	Fast convergence to local optima for smooth problems.	Deterministic iterates
		Gradient-based
		Automatic stochastic start points
		Removes many start points heuristically
`MultiStart`	Fast convergence to local optima for smooth problems.	Deterministic iterates
		Can run in parallel; see "How to Use Parallel Processing" on page 8-11
		Gradient-based
		Stochastic or deterministic start points, or combination of both
		Automatic stochastic start points
		Runs all start points
		Choice of local solver: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`
`patternsearch`	Proven convergence to local optimum, slower than gradient-based solvers.	Deterministic iterates
		Can run in parallel; see "How to Use Parallel Processing" on page 8-11
		No gradients
		User-supplied start point

Solver	Convergence	Characteristics
`ga`	No convergence proof.	Stochastic iterates
		Can run in parallel; see "How to Use Parallel Processing" on page 8-11
		Population-based
		No gradients
		Automatic start population, or user-supplied population, or combination of both
`simulannealbnd`	Proven to converge to global optimum for bounded problems with very slow cooling schedule.	Stochastic iterates
		No gradients
		User-supplied start point
		Only bound constraints

Explanation of some characteristics:

- Convergence — Solvers can fail to converge to any solution when started far from a local minimum. When started near a local minimum, gradient-based solvers converge to a local minimum quickly for smooth problems. `patternsearch` provably converges for a wide range of problems, but the convergence is slower than gradient-based solvers. Both `ga` and `simulannealbnd` can fail to converge in a reasonable amount of time for some problems, although they are often effective.

- Iterates — Solvers iterate to find solutions. The steps in the iteration are iterates. Some solvers have deterministic iterates. Others use random numbers and have stochastic iterates.

- Gradients — Some solvers use estimated or user-supplied derivatives in calculating the iterates. Other solvers do not use or estimate derivatives, but use only objective and constraint function values.

- Start points — Most solvers require you to provide a starting point for the optimization. One reason they require a start point is to obtain the dimension of the decision variables. `ga` does not require any starting points, because it takes the dimension of the decision variables as an input. `ga` can generate its start population automatically.

Compare the characteristics of Global Optimization Toolbox solvers to Optimization Toolbox solvers.

Solver	Convergence	Characteristics
`fmincon, fminunc, fseminf, lsqcurvefit, lsqnonlin`	Proven quadratic convergence to local optima for smooth problems	Deterministic iterates
		Gradient-based
		User-supplied starting point
`fminsearch`	No convergence proof — counterexamples exist.	Deterministic iterates
		No gradients
		User-supplied start point
`fminbnd`	Proven convergence to local optima for smooth problems, slower than quadratic.	Deterministic iterates
		No gradients
		User-supplied start point
		Only one-dimensional problems

All these Optimization Toolbox solvers:

- Have deterministic iterates
- Start from one user-supplied point
- Search just one basin of attraction

Why Are Some Solvers Objects?

GlobalSearch and MultiStart are objects. What does this mean for you?

- You create a GlobalSearch or MultiStart object before running your problem.
- You can reuse the object for running multiple problems.
- GlobalSearch and MultiStart objects are containers for algorithms and global options. You use these objects to run a local solver multiple times. The local solver has its own options.

For more information, see the *Object-Oriented Programming* documentation.

Writing Files for Optimization Functions

Computing Objective Functions

In this section...

Objective (Fitness) Functions

To use Global Optimization Toolbox functions, you must first write a file (or an anonymous function) that computes the function you want to optimize. This function is called an objective function for most solvers or a fitness function for `ga`. The function should accept a vector whose length is the number of independent variables, and should return a scalar. For vectorized solvers, the function should accept a matrix (where each row represents one input vector), and return a vector of objective function values. This section shows how to write the file.

Example: Writing a Function File

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

$$f(x) = x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2.$$

The 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 file, do the following steps:

1 Select **New > Script** (**Ctrl+N**) from the MATLAB **File** menu. This opens a new file in the editor.

2 In the 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) + 4*x(2)^2 - 3*x(2);
```

3 Save the file in a folder on the MATLAB path.

To check that the file returns the correct value, enter

```
my_fun([2 3])

ans =
    31
```

Example: Writing a Vectorized Function

The `ga` and `patternsearch` solvers optionally compute the objective functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

To compute in vectorized fashion:

- Write your objective function to:
  - Accept a matrix with an arbitrary number of rows
  - Return the vector of function values of each row
- If you have a nonlinear constraint, be sure to write the constraint in a vectorized fashion. For details, see "Vectorized Constraints" on page 2-7.
- Set the `Vectorized` option to `'on'` with `gaoptimset` or `psoptimset`, or set **User function evaluation > Evaluate objective/fitness and constraint functions** to `vectorized` in the Optimization Tool. For `patternsearch`, also set `CompletePoll` to `'on'`. Be sure to pass the options structure to the solver.

For example, to write the objective function of "Example: Writing a Function File" on page 2-2 in a vectorized fashion,

```
function z = my_fun(x)
z = x(:,1).^2 - 2*x(:,1).*x(:,2) + 6*x(:,1) + ...
    4*x(:,2).^2 - 3*x(:,2);
```

To use `my_fun` as a vectorized objective function for `patternsearch`:

```
options = psoptimset('CompletePoll','on','Vectorized','on');
[x fval] = patternsearch(@my_fun,[1 1],[],[],[],[],[],[],...
    [],options);
```

To use my_fun as a vectorized objective function for ga:

```
options = gaoptimset('Vectorized','on');
[x fval] = ga(@my_fun,2,[],[],[],[],[],[],[],options);
```

For more information on writing vectorized functions for patternsearch, see "Vectorizing the Objective and Constraint Functions" on page 4-72. For more information on writing vectorized functions for ga, see "Vectorizing the Fitness Function" on page 5-81.

Gradients and Hessians

If you use GlobalSearch or MultiStart, your objective function can return derivatives (gradient, Jacobian, or Hessian). For details on how to include this syntax in your objective function, see "Writing Objective Functions" in Optimization Toolbox documentation. Use optimset to set options so that your solver uses the derivative information:

Local Solver = fmincon, fminunc

Condition	Option Setting
Objective function contains gradient	'GradObj' = 'on'
Objective function contains Hessian	'Hessian' = 'on'
Constraint function contains gradient	'GradConstr' = 'on'
Calculate Hessians of Lagrangian in an extra function	'Hessian' = 'on', 'HessFcn' = function handle

For more information about Hessians for fmincon, see "Hessian".

Local Solver = lsqcurvefit, lsqnonlin

Condition	Option Setting
Objective function contains Jacobian	'Jacobian' = 'on'

Maximizing vs. Minimizing

Global Optimization 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 $f(x)$, minimize $-f(x)$, 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

$$f(x) = x_1^2 - 2x_1x_2 + 6x_1 + 4x_2^2 - 3x_2.$$

Write your function file to compute

$$g(x) = -f(x) = -x_1^2 + 2x_1x_2 - 6x_1 - 4x_2^2 + 3x_2,$$

and minimize $g(x)$.

Constraints

Many Global Optimization Toolbox functions accept bounds, linear constraints, or nonlinear constraints. To see how to include these constraints in your problem, see "Writing Constraints" in the Optimization Toolbox documentation. Try consulting these pertinent links to sections:

- "Bound Constraints"
- "Linear Inequality Constraints" (linear equality constraints have the same form)
- "Nonlinear Constraints"

Set Bounds

It is more important to set bounds for global solvers than for local solvers. Global solvers use bounds in a variety of ways:

- `GlobalSearch` requires bounds for its scatter-search point generation. If you do not provide bounds, `GlobalSearch` bounds each component below by `-9999` and above by `10001`. However, these bounds can easily be inappropriate.

- If you do not provide bounds and do not provide custom start points, `MultiStart` bounds each component below by `-1000` and above by `1000`. However, these bounds can easily be inappropriate.

- `ga` uses bounds and linear constraints for its initial population generation. For unbounded problems, `ga` uses a default of `0` as the lower bound and `1` as the upper bound for each dimension for initial point generation. For bounded problems, and problems with linear constraints, `ga` uses the bounds and constraints to make the initial population.

- `simulannealbnd` and `patternsearch` do not require bounds, although they can use bounds.

Gradients and Hessians

If you use `GlobalSearch` or `MultiStart` with `fmincon`, your nonlinear constraint functions can return derivatives (gradient or Hessian). For details, see "Gradients and Hessians" on page 2-4.

Vectorized Constraints

The `ga` and `patternsearch` solvers optionally compute the nonlinear constraint functions of a collection of vectors in one function call. This method can take less time than computing the objective functions of the vectors serially. This method is called a vectorized function call.

For the solver to compute in a vectorized manner, you must vectorize both your objective (fitness) function and nonlinear constraint function. For details, see "Vectorizing the Objective and Constraint Functions" on page 4-72.

As an example, suppose your nonlinear constraints for a three-dimensional problem are

$$\frac{x_1^2}{4} + \frac{x_2^2}{9} + \frac{x_3^2}{25} \leq 6$$
$$x_3 \geq \cosh\left(x_1 + x_2\right)$$
$$x_1 x_2 x_3 = 2.$$

The following code gives these nonlinear constraints in a vectorized fashion, assuming that the rows of your input matrix `x` are your population or input vectors:

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

c(:,1) = x(:,1).^2/4 + x(:,2).^2/9 + x(:,3).^2/25 - 6;
c(:,2) = cosh(x(:,1) + x(:,2)) - x(:,3);
ceq = x(:,1).*x(:,2).*x(:,3) - 2;
```

For example, minimize the vectorized quadratic function

```
function y = vfun(x)
y = -x(:,1).^2 - x(:,2).^2 - x(:,3).^2;
```

over the region with constraints `nlinconst` using `patternsearch`:

```
options = psoptimset('CompletePoll','on','Vectorized','on');
[x fval] = patternsearch(@vfun,[1,1,2],[],[],[],[],[],[],...
    @nlinconst,options)
Optimization terminated: mesh size less than options.TolMesh
```

```
      and constraint violation is less than options.TolCon.

  x =
      0.2191    0.7500   12.1712

  fval =
   -148.7480
```

Using ga:

```
  options = gaoptimset('Vectorized','on');
  [x fval] = ga(@vfun,3,[],[],[],[],[],[],@nlinconst,options)
  Optimization terminated: maximum number of generations exceeded.

  x =
     -1.4098   -0.1216   11.6664

  fval =
   -138.1066
```

For this problem patternsearch computes the solution far more quickly and accurately.

3

Using GlobalSearch and MultiStart

How to Optimize with GlobalSearch and MultiStart

In this section...
"Problems You Can Solve with GlobalSearch and MultiStart" on page 3-2
"Outline of Steps" on page 3-2
"Determining Inputs for the Problem" on page 3-4
"Create a Problem Structure" on page 3-4
"Create a Solver Object" on page 3-13
"Set Start Points for MultiStart" on page 3-16
"Run the Solver" on page 3-19

Problems You Can Solve with GlobalSearch and MultiStart

The GlobalSearch and MultiStart solvers apply to problems with smooth objective and constraint functions. The solvers search for a global minimum, or for a set of local minima. For more information on which solver to use, see "Choosing a Solver" on page 1-17.

GlobalSearch and MultiStart work by starting a local solver, such as fmincon, from a variety of start points. Generally the start points are random. However, for MultiStart you can provide a set of start points. For more information, see "How GlobalSearch and MultiStart Work" on page 3-35.

To find out how to use these solvers, see "Outline of Steps" on page 3-2. For examples using these solvers, see the example index.

Outline of Steps

To find a global or multiple local solutions:

1 "Create a Problem Structure" on page 3-4

2 "Create a Solver Object" on page 3-13

3 (Optional, MultiStart only) "Set Start Points for MultiStart" on page 3-16

4 "Run the Solver" on page 3-19

The following figure illustrates these steps.

Information you have	Local Solver X0 Objective Constraints Local Options	Global options	Optional, MultiStart only Start Points
Command to use	createOptimProblem	GlobalSearch or MultiStart	RandomStartPointSet or CustomStartPointSet
Resulting Object or Structure	Problem Structure	Solver Object	Start Points Object

run

Results
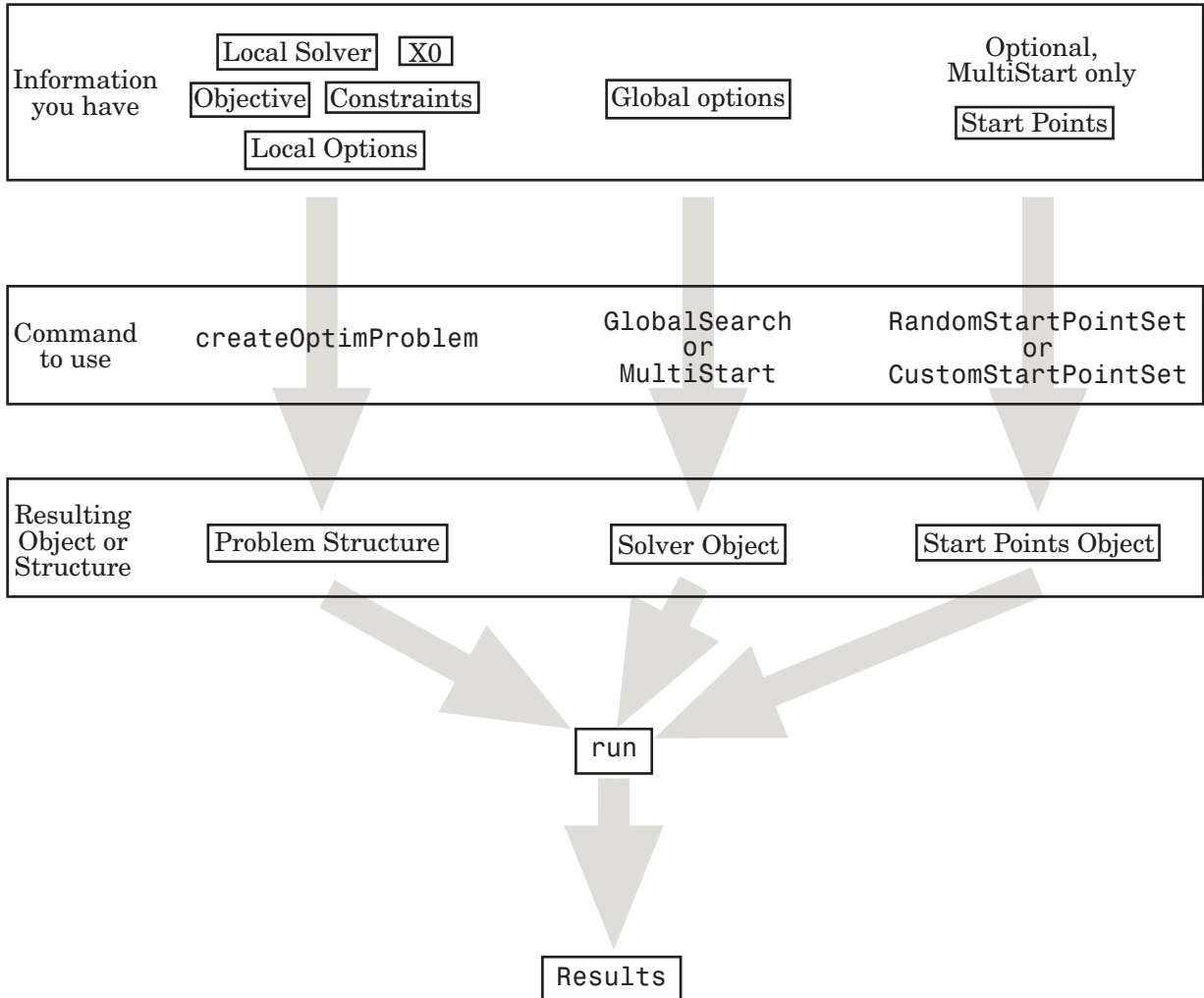
Determining Inputs for the Problem

A problem structure defines a local optimization problem using a local solver, such as fmincon. Therefore, most of the documentation on choosing a solver or preparing the inputs is in the Optimization Toolbox documentation.

Required Inputs

Input	Link to Documentation
Local solver	"Optimization Decision Table" in the Optimization Toolbox documentation.
Objective function	"Computing Objective Functions" on page 2-2
Start point x0	

Optional Inputs

Input	Link to Documentation
Constraint functions	"Constraints" on page 2-6
Local options structure	"Setting Options"

Create a Problem Structure

To use the GlobalSearch or MultiStart solvers, you must first create a problem structure. There are two ways to create a problem structure:

- "Using the createOptimProblem Function" on page 3-4
- "Exporting from the Optimization Tool" on page 3-8

Using the createOptimProblem Function

Follow these steps to create a problem structure using the createOptimProblem function.

1 Define your objective function as a file or anonymous function. For details, see "Computing Objective Functions" on page 2-2. If your solver

is lsqcurvefit or lsqnonlin, ensure the objective function returns a vector, not scalar.

2 If relevant, create your constraints, such as bounds and nonlinear constraint functions. For details, see "Constraints" on page 2-6.

3 Create a start point. For example, to create a three-dimensional random start point xstart:

```
xstart = randn(3,1);
```

4 (Optional) Create an options structure using optimset. For example,

```
options = optimset('Algorithm','interior-point');
```

5 Enter

```
problem = createOptimProblem(solver,
```

where *solver* is the name of your local solver:

- For GlobalSearch: 'fmincon'
- For MultiStart the choices are:
  - 'fmincon'
  - 'fminunc'
  - 'lsqcurvefit'
  - 'lsqnonlin'

  For help choosing, see "Optimization Decision Table".

6 Set an initial point using the 'x0' parameter. If your initial point is xstart, and your solver is fmincon, your entry is now

```
problem = createOptimProblem('fmincon','x0',xstart,
```

7 Include the function handle for your objective function in objective:

```
problem = createOptimProblem('fmincon','x0',xstart, ...
    'objective',@objfun,
```

8 Set bounds and other constraints as applicable.

Constraint	Name
lower bounds	`'lb'`
upper bounds	`'ub'`
matrix `Aineq` for linear inequalities `Aineq x ≤ bineq`	`'Aineq'`
vector `bineq` for linear inequalities `Aineq x ≤ bineq`	`'bineq'`
matrix `Aeq` for linear equalities `Aeq x = beq`	`'Aeq'`
vector `beq` for linear equalities `Aeq x = beq`	`'beq'`
nonlinear constraint function	`'nonlcon'`

9 If using the `lsqcurvefit` local solver, include vectors of input data and response data, named `'xdata'` and `'ydata'` respectively.

10 *Best practice: validate the problem structure by running your solver on the structure.* For example, if your local solver is `fmincon`:

```
[x fval eflag output] = fmincon(problem);
```

Note Specify `fmincon` as the solver for `GlobalSearch`, even if you have no constraints. However, you cannot run `fmincon` on a problem without constraints. Add an artificial constraint to the problem to validate the structure:

```
problem.lb = -Inf;
```

Example: Creating a Problem Structure with createOptimProblem.
This example minimizes the function from "Run the Solver" on page 3-19, subject to the constraint $x_1 + 2x_2 \geq 4$. The objective is

$$\text{sixmin} = 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4.$$

Use the `interior-point` algorithm of `fmincon`, and set the start point to `[2;3]`.

1 Write a function handle for the objective function.

```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
```

2 Write the linear constraint matrices. Change the constraint to "less than" form:

```
A = [-1,-2];
b = -4;
```

3 Create the local options structure to use the `interior-point` algorithm:

```
opts = optimset('Algorithm','interior-point');
```

4 Create the problem structure with `createOptimProblem`:

```
problem = createOptimProblem('fmincon', ...
    'x0',[2;3],'objective',sixmin, ...
    'Aineq',A,'bineq',b,'options',opts);
```

5 The resulting structure:

```
problem =
    objective: @(x)(4*x(1)^2-2.1*x(1)^4+x(1)^6/3+ ...
                  x(1)*x(2)-4*x(2)^2+4*x(2)^4)
           x0: [2x1 double]
        Aineq: [-1 -2]
        bineq: -4
          Aeq: []
          beq: []
           lb: []
           ub: []
      nonlcon: []
       solver: 'fmincon'
      options: [1x1 struct]
```

6 *Best practice: validate the problem structure by running your solver on the structure:*

```
[x fval eflag output] = fmincon(problem);
```

> **Note** Specify fmincon as the solver for GlobalSearch, even if you have
> no constraints. However, you cannot run fmincon on a problem without
> constraints. Add an artificial constraint to the problem to validate the
> structure:
>
>   problem.lb = -Inf;

Exporting from the Optimization Tool

Follow these steps to create a problem structure using the Optimization Tool.

1 Define your objective function as a file or anonymous function. For
details, see "Computing Objective Functions" on page 2-2. If your solver
is lsqcurvefit or lsqnonlin, ensure the objective function returns a
vector, not scalar.

2 If relevant, create nonlinear constraint functions. For details, see
"Nonlinear Constraints".

3 Create a start point. For example, to create a three-dimensional random
start point xstart:

    xstart = randn(3,1);

4 Open the Optimization Tool by entering optimtool at the command line, or
by choosing Start > Toolboxes > Optimization > Optimization Tool.

5 Choose the local **Solver**.
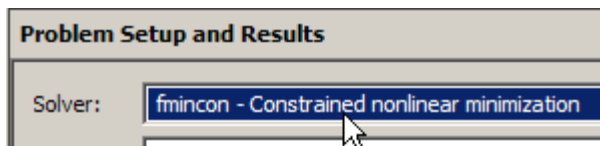
- For GlobalSearch: fmincon (default).
- For MultiStart:
  - fmincon (default)

- – `fminunc`
- – `lsqcurvefit`
- – `lsqnonlin`

For help choosing, see "Optimization Decision Table".

6 Choose an appropriate **Algorithm**. For help choosing, see "Choosing the Algorithm".
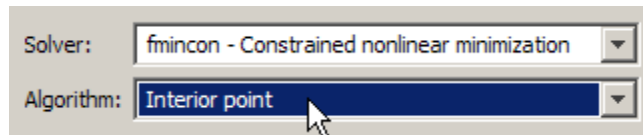
7 Set an initial point (**Start point**).

8 Include the function handle for your objective function in **Objective function**, and, if applicable, include your **Nonlinear constraint function**. For example,
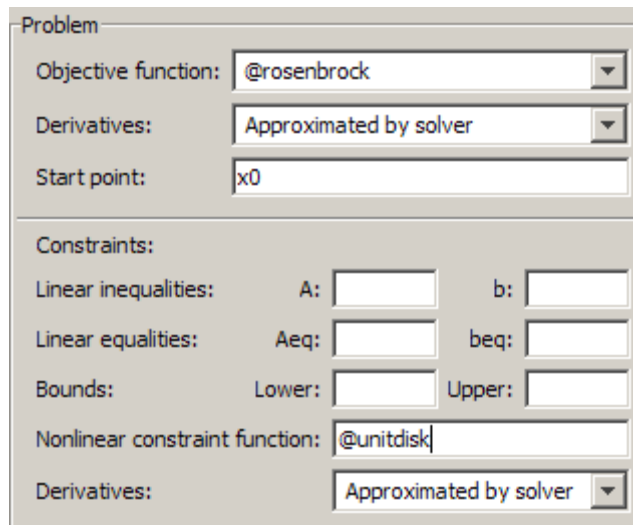
9 Set bounds, linear constraints, or local **Options**. For details on constraints, see "Writing Constraints".

10 *Best practice: run the problem to verify the setup.*
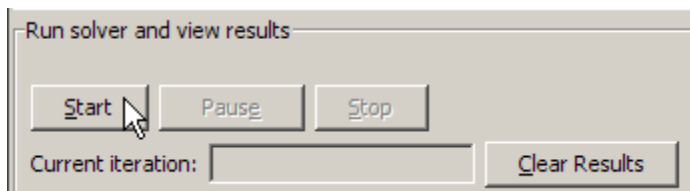
Note You must specify fmincon as the solver for GlobalSearch, even if you have no constraints. However, you cannot run fmincon on a problem without constraints. Add an artificial constraint to the problem to verify the setup.
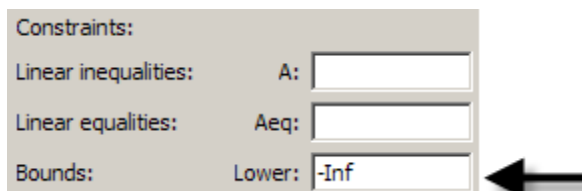
11 Choose **File > Export to Workspace** and select **Export problem and options to a MATLAB structure named**
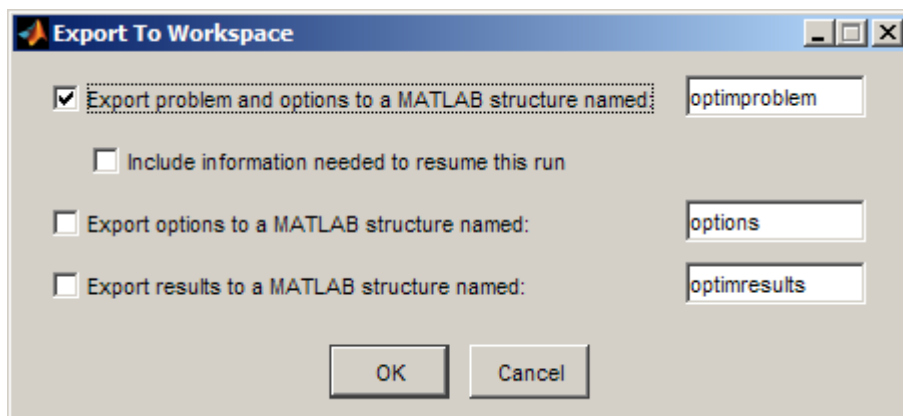
Example: Creating a Problem Structure with the Optimization Tool.
This example minimizes the function from "Run the Solver" on page 3-19,
subject to the constraint $x_1 + 2x_2 \geq 4$. The objective is

$$\text{sixmin} = 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4.$$

Use the `interior-point` algorithm of `fmincon`, and set the start point to
`[2;3]`.

1 Write a function handle for the objective function.

```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
```
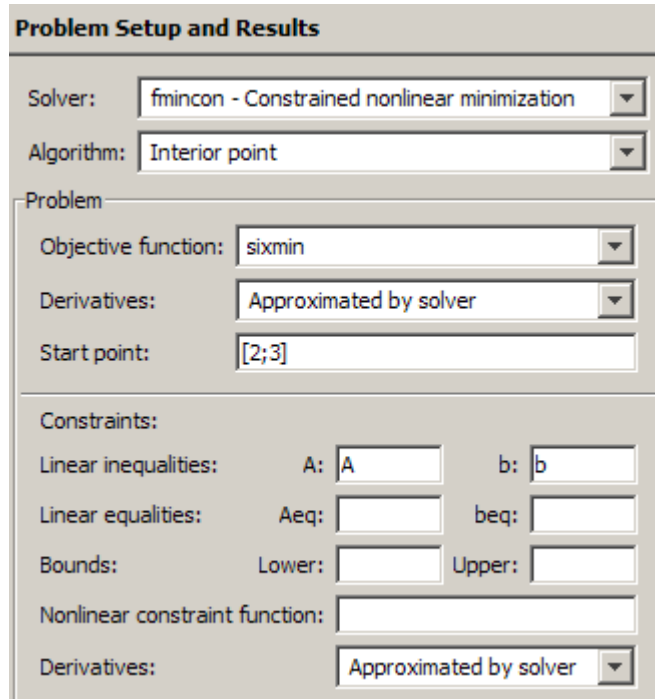
2 Write the linear constraint matrices. Change the constraint to "less than"
form:

```
A = [-1,-2];
b = -4;
```
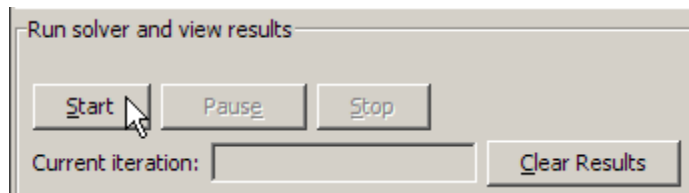
3 Launch the Optimization Tool by entering `optimtool` at the MATLAB
command line.

4 Set the solver, algorithm, objective, start point, and constraints.

Problem Setup and Results

Solver: fmincon - Constrained nonlinear minimization ▼

Algorithm: Interior point   ▼

Problem
  Objective function: sixmin   ▼

  Derivatives: Approximated by solver   ▼

  Start point: [2;3]

  Constraints:

  Linear inequalities: A: A   b: b

  Linear equalities: Aeq: beq:

  Bounds: Lower: Upper:

  Nonlinear constraint function:

  Derivatives: Approximated by solver   ▼

5 *Best practice: run the problem to verify the setup.*

Run solver and view results

    Start   Pause   Stop

Current iteration: Clear Results

The problem runs successfully.
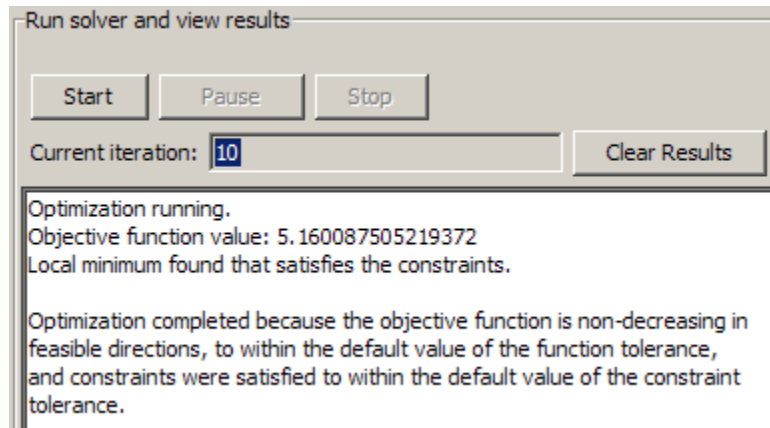
6 Choose **File > Export to Workspace** and select **Export problem and options to a MATLAB structure named**
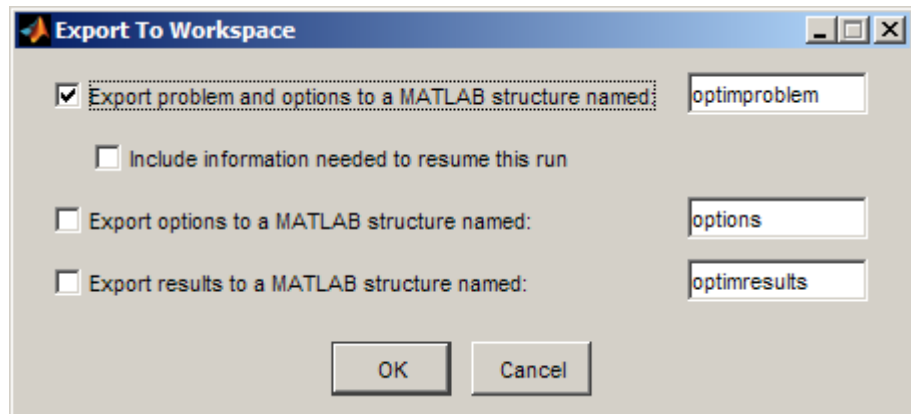
Create a Solver Object

A solver object contains your preferences for the global portion of the optimization.

You do not need to set any preferences. Create a GlobalSearch object named gs with default settings as follows:

```
gs = GlobalSearch;
```

Similarly, create a MultiStart object named ms with default settings as follows:

```
ms = MultiStart;
```

Properties (Global Options) of Solver Objects

Global options are properties of a GlobalSearch or MultiStart object.

Properties for both GlobalSearch and MultiStart

Property Name	Meaning
Display	Detail level of iterative display. Set to 'off' for no display, 'final' (default) for a report at the end of the run, or 'iter' for reports as the solver progresses. For more information and examples, see "Iterative Display" on page 3-28.
TolFun	Solvers consider objective function values within TolFun of each other to be identical (not distinct). Default: 1e-6. Solvers group solutions when the solutions satisfy both TolFun and TolX tolerances.
TolX	Solvers consider solutions within TolX distance of each other to be identical (not distinct). Default: 1e-6. Solvers group solutions when the solutions satisfy both TolFun and TolX tolerances.
MaxTime	Solvers halt if the run exceeds MaxTime seconds, as measured by a clock (not processor seconds). Default: Inf
StartPointsToRun	Choose whether to run 'all' (default) start points, only those points that satisfy 'bounds', or only those points that are feasible with respect to bounds and inequality constraints with 'bounds-ineqs'. For an example, see "Example: Using Only Feasible Start Points" on page 3-70.

Properties for GlobalSearch

Property Name	Meaning
NumTrialPoints	Number of trial points to use examine. Default: 1000
BasinRadiusFactor	See "Properties" on page 12-30 for detailed descriptions of these properties.
DistanceThresholdFactor	
MaxWaitCycle	
NumStageOnePoints	
PenaltyThresholdFactor	

Properties for MultiStart

Property Name	Meaning
UseParallel	When 'always', MultiStart attempts to distribute start points to multiple processors for the local solver. Disable by setting to 'never' (default). For details, see "How to Use Parallel Processing" on page 8-11. For an example, see "Example: Parallel MultiStart" on page 3-74.

Example: Creating a Nondefault GlobalSearch Object

Suppose you want to solve a problem and:

- Consider local solutions identical if they are within 0.01 of each other and the function values are within the default TolFun tolerance.

- Spend no more than 2000 seconds on the computation.

To solve the problem, create a GlobalSearch object gs as follows:

```
gs = GlobalSearch('TolX',0.01,'MaxTime',2000);
```

Example: Creating a Nondefault MultiStart Object

Suppose you want to solve a problem such that:

- You consider local solutions identical if they are within 0.01 of each other and the function values are within the default `TolFun` tolerance.

- You spend no more than 2000 seconds on the computation.

To solve the problem, create a `MultiStart` object `ms` as follows:

```
ms = MultiStart('TolX',0.01,'MaxTime',2000);
```

Set Start Points for MultiStart

There are four ways you tell `MultiStart` which start points to use for the local solver:

- Pass a positive integer `k`. `MultiStart` generates `k - 1` start points as if using a `RandomStartPointSet` object and the `problem` structure. `MultiStart` also uses the `x0` start point from the `problem` structure, for a total of `k` start points.

- Pass a `RandomStartPointSet` object.

- Pass a `CustomStartPointSet` object.

- Pass a cell array of `RandomStartPointSet` and `CustomStartPointSet` objects. Pass a cell array if you have some specific points you want to run, but also want `MultiStart` to use other random start points.

Note You can control whether `MultiStart` uses all start points, or only those points that satisfy bounds or other inequality constraints. For more information, see "Filter Start Points (Optional)" on page 3-42.

Positive Integer for Start Points

The syntax for running `MultiStart` for `k` start points is

```
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,k);
```

The positive integer `k` specifies the number of start points `MultiStart` uses. `MultiStart` generates random start points using the dimension of the problem and bounds from the `problem` structure. `MultiStart` generates `k - 1` random start points, and also uses the `x0` start point from the `problem` structure.

RandomStartPointSet Object for Start Points

Create a RandomStartPointSet object as follows:

```
stpoints = RandomStartPointSet;
```

By default a RandomStartPointSet object generates 10 start points. Control the number of start points with the NumStartPoints property. For example, to generate 40 start points:

```
stpoints = RandomStartPointSet('NumStartPoints',40);
```

You can set an ArtificialBound for a RandomStartPointSet.

- If a component has no bounds, RandomStartPointSet uses a lower bound of -ArtificialBound, and an upper bound of ArtificialBound.

- If a component has a lower bound lb but no upper bound, RandomStartPointSet uses an upper bound of lb + 2*ArtificialBound.

- Similarly, if a component has an upper bound ub but no lower bound, RandomStartPointSet uses a lower bound of ub - 2*ArtificialBound.

For example, to generate 100 start points with an ArtificialBound of 50:

```
stpoints = RandomStartPointSet('NumStartPoints',100, ...
    'ArtificialBound',50);
```

CustomStartPointSet Object for Start Points

To use a specific set of starting points, package them in a CustomStartPointSet as follows:

1 Place the starting points in a matrix. Each row of the matrix represents one starting point. MultiStart runs all the rows of the matrix, subject to filtering with the StartPointsToRun property. For more information, see "MultiStart Algorithm" on page 3-41.

2 Create a CustomStartPointSet object from the matrix:

```
tpoints = CustomStartPointSet(ptmatrix);
```

For example, create a set of 40 five-dimensional points, with each component of a point equal to 10 plus an exponentially distributed variable with mean 25:

```
pts = -25*log(rand(40,5)) + 10;
tpoints = CustomStartPointSet(pts);
```

To get the original matrix of points from a CustomStartPointSet object, use the list method:

```
pts = list(tpoints); % Assumes tpoints is a CustomStartPointSet
```

A CustomStartPointSet has two properties: DimStartPoints and NumStartPoints. You can use these properties to query a CustomStartPointSet object. For example, the tpoints object in the example has the following properties:

```
tpoints.DimStartPoints
ans =
     5

tpoints.NumStartPoints
ans =
    40
```

Cell Array of Objects for Start Points

To use a specific set of starting points along with some randomly generated points, pass a cell array of RandomStartPointSet or CustomStartPointSet objects.

For example, to use both the 40 specific five-dimensional points of "CustomStartPointSet Object for Start Points" on page 3-17 and 40 additional five-dimensional points from RandomStartPointSet:

```
pts = -25*log(rand(40,5)) + 10;
tpoints = CustomStartPointSet(pts);
rpts = RandomStartPointSet('NumStartPoints',40);
allpts = {tpoints,rpts};
```

Run MultiStart with the allpts cell array:

```
% Assume ms and problem exist
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,allpts);
```

Run the Solver

Running a solver is nearly identical for `GlobalSearch` and `MultiStart`. The only difference in syntax is `MultiStart` takes an additional input describing the start points.

For example, suppose you want to find several local minima of the `sixmin` function

$$\text{sixmin} = 4x^2 - 2.1x^4 + x^6/3 + xy - 4y^2 + 4y^4.$$
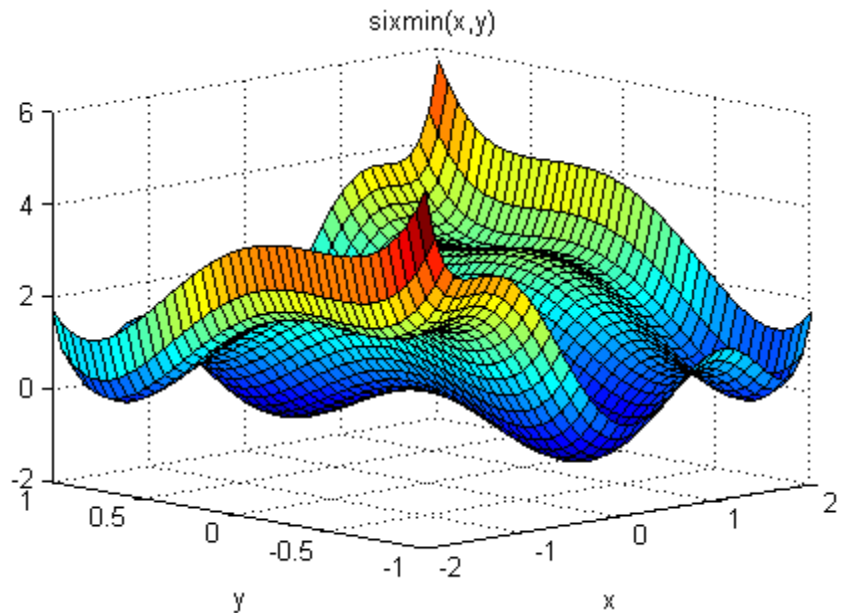
This function is also called the six-hump camel back function [3]. All the local minima lie in the region $-3 \le x, y \le 3$.

Example of Run with GlobalSearch

To find several local minima of the problem described in "Run the Solver" on page 3-19 using `GlobalSearch`, enter:

```
gs = GlobalSearch;
opts = optimset('Algorithm','interior-point');
```

3-19

```
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

The output of the run (which varies, based on the random seed):

```
xming,fming,flagg,outptg,manyminsg
xming =
   -0.0898    0.7127

fming =
   -1.0316

flagg =
     1

outptg =
                 funcCount: 2245
           localSolverTotal: 8
         localSolverSuccess: 8
      localSolverIncomplete: 0
      localSolverNoSolution: 0
                   message: [1x137 char]

manyminsg =
  1x4 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

Example of Run with MultiStart

To find several local minima of the problem described in "Run the Solver" on page 3-19 using 50 runs of fmincon with MultiStart, enter:

```
% % Set the default stream to get exactly the same output
% s = RandStream('mt19937ar','Seed',14);
% RandStream.setDefaultStream(s);
ms = MultiStart;
opts = optimset('Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);
```

The output of the run (which varies based on the random seed):

```
xminm,fminm,flagm,outptm,manyminsm

xminm =
   -0.0898    0.7127

fminm =
   -1.0316

flagm =
     1

outptm =
                 funcCount: 2035
           localSolverTotal: 50
         localSolverSuccess: 50
      localSolverIncomplete: 0
      localSolverNoSolution: 0
                    message: [1x128 char]

manyminsm =
  1x6 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
```

```
Output
XO
```

In this case, `MultiStart` located all six local minima, while `GlobalSearch` located four. For pictures of the `MultiStart` solutions, see "Example: Visualizing the Basins of Attraction" on page 3-32.

Examining Results

Single Solution

You obtain the single best solution found during the run by calling `run` with the syntax

```
[x fval eflag output] = run(...);
```

- `x` is the location of the local minimum with smallest objective function value.

- `fval` is the objective function value evaluated at `x`.

- `eflag` is an exit flag for the global solver. Values:

Global Solver Exit Flags

2	At least one local minimum found. Some runs of the local solver converged (had positive exit flag).
1	At least one local minimum found. All runs of the local solver converged (had positive exit flag).
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the `MaxIter` or `MaxFunEvals` tolerances.
-1	One or more local solver runs stopped by the local solver output function or plot function.
-2	No feasible local minimum found.
-5	`MaxTime` limit exceeded.

Global Solver Exit Flags (Continued)

-8	No solution found. All runs had local solver exit flag -2 or smaller, not all equal -2.
-10	Failures encountered in user-provided functions.

- output is a structure with details about the multiple runs of the local solver. For more information, see "Global Output Structures" on page 3-31.

The list of outputs is for the case eflag > 0. If eflag <= 0, then x is the following:

- If some local solutions are feasible, x represents the location of the lowest objective function value. "Feasible" means the constraint violations are smaller than problem.options.TolCon.

- If no solutions are feasible, x is the solution with lowest infeasibility.

- If no solutions exist, x, fval, and output are empty ([ ]).

Multiple Solutions

You obtain multiple solutions in an object by calling run with the syntax

```
[x fval eflag output manymins] = run(...);
```

manymins is a vector of solution objects; see GlobalOptimSolution. The vector is in order of objective function value, from lowest (best) to highest (worst). Each solution object contains the following properties (fields):

- X — a local minimum

- Fval — the value of the objective function at X

- Exitflag — the exit flag for the global solver (described here)

- Output — an output structure (described here)

- X0 — a cell array of start points that led to the solution point X

There are several ways to examine the vector of solution objects:

- In the MATLAB Workspace Browser. Double-click the solution object, and then double-click the resulting display in the Variable Editor.

- Using dot addressing. GlobalOptimSolution properties are capitalized. Use proper capitalization to access the properties.

For example, to find the vector of function values, enter:

```
fcnvals = [manymins.Fval]

fcnvals =
```

```
       -1.0316    -0.2155            0
```

To get a cell array of all the start points that led to the lowest function value (the first element of manymins), enter:

```
smallXO = manymins(1).XO
```

- Plot some field values. For example, to see the range of resulting Fval, enter:

```
hist([manymins.Fval])
```

This results in a histogram of the computed function values. (The figure shows a histogram from a different example than the previous few figures.)
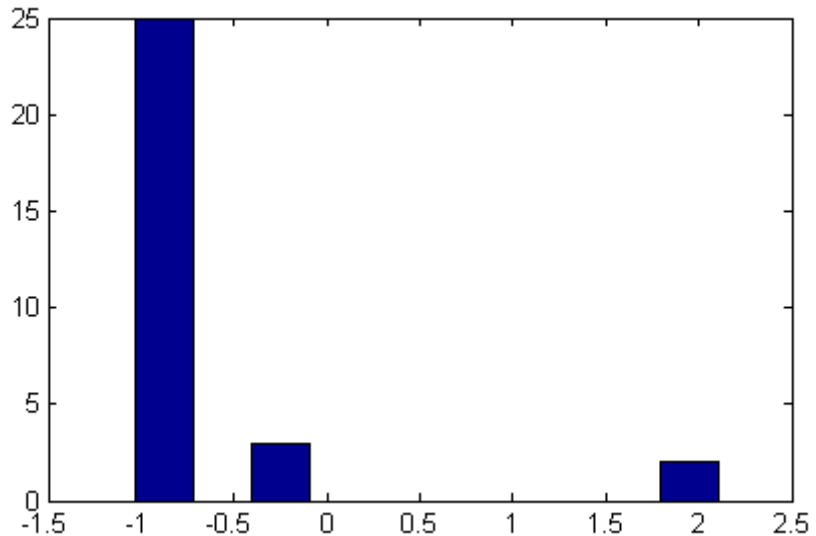
Example: Changing the Definition of Distinct Solutions

You might find out, after obtaining multiple local solutions, that your tolerances were not appropriate. You can have many more local solutions than you want, spaced too closely together. Or you can have fewer solutions

than you want, with `GlobalSearch` or `MultiStart` clumping together too many solutions.

To deal with this situation, run the solver again with different tolerances. The `TolX` and `TolFun` tolerances determine how the solvers group their outputs into the `GlobalOptimSolution` vector. These tolerances are properties of the `GlobalSearch` or `MultiStart` object.

For example, suppose you want to use the `active-set` algorithm in `fmincon` to solve the problem in "Example of Run with MultiStart" on page 3-20. Further suppose that you want to have tolerances of `0.01` for both `TolX` and `TolFun`. The `run` method groups local solutions whose objective function values are within `TolFun` of each other, and which are also less than `TolX` apart from each other. To obtain the solution:

```
ms = MultiStart('TolFun',0.01,'TolX',0.01);
opts = optimset('Algorithm','active-set');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xminm,fminm,flagm,outptm,someminsm] = run(ms,problem,50);

MultiStart completed the runs from all start points.

All 50 local solver runs converged with a
positive local solver exit flag.

someminsm

someminsm =

  1x6 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
```

```
X0
```

In this case, `MultiStart` generated six distinct solutions. Here "distinct" means that the solutions are more than 0.01 apart in either objective function value or location.

Iterative Display

Iterative display gives you information about the progress of solvers during their runs.

There are two types of iterative display:

- Global solver display
- Local solver display

Both types appear at the command line, depending on global and local options.

Obtain local solver iterative display by setting the `Display` option in the `problem.options` structure to `'iter'` or `'iter-detailed'` with `optimset`. For more information, see "Displaying Iterative Output" in the Optimization Toolbox documentation.

Obtain global solver iterative display by setting the `Display` property in the `GlobalSearch` or `MultiStart` object to `'iter'`.

Global solvers set the default `Display` option of the local solver to `'off'`, unless the problem structure has a value for this option. Global solvers do not override any setting you make for local options.

Note Setting the local solver `Display` option to anything other than `'off'` can produce a great deal of output. The default `Display` option created by `optimset(@`*solver*`)` is `'final'`.

Example: Types of Iterative Display

Run the example described in "Run the Solver" on page 3-19 using `GlobalSearch` with `GlobalSearch` iterative display:

```
gs = GlobalSearch('Display','iter');
opts = optimset('Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
    'options',opts);
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	34	-1.032			-1.032	1	Initial Point
200	1441	-1.032			0	1	Stage 1 Local
300	1641	-1.032	290.7	0.07006			Stage 2 Search
400	1841	-1.032	101	0.8017			Stage 2 Search
500	2041	-1.032	51.23	3.483			Stage 2 Search
541	2157	-1.032	5.348	5.456	-1.032	1	Stage 2 Local
542	2190	-1.032	2.993	5.348	-1.032	1	Stage 2 Local
544	2227	-1.032	1.807	2.993	-1.032	1	Stage 2 Local
545	2260	-1.032	1.609	1.807	-1.032	1	Stage 2 Local
546	2296	-1.032	0.9061	1.609	-1.032	1	Stage 2 Local
552	2348	-1.032	0.6832	0.9061	-0.2155	1	Stage 2 Local
600	2444	-1.032	1.079	0.1235			Stage 2 Search
700	2644	-1.032	0.4859	-0.4791			Stage 2 Search
800	2844	-1.032	136.9	0.8202			Stage 2 Search
900	3044	-1.032	37.77	0.4234			Stage 2 Search
1000	3244	-1.032	-0.05542	-0.598			Stage 2 Search

```
GlobalSearch stopped because it analyzed all the start points.

All 8 local solver runs converged with a positive local solver exit flag.
```

Run the same example without `GlobalSearch` iterative display, but with `fmincon` iterative display:

```
gs.Display = 'final';
problem.options.Display = 'iter';
[xming,fming,flagg,outptg,manyminsg] = run(gs,problem);
```

				First-order	Norm of
Iter	F-count	f(x)	Feasibility	optimality	step
0	3	4.823333e+001	0.000e+000	1.088e+002	
1	7	2.020476e+000	0.000e+000	2.176e+000	2.488e+000
2	10	6.525252e-001	0.000e+000	1.937e+000	1.886e+000
3	13	-8.776121e-001	0.000e+000	9.076e-001	8.539e-001
4	16	-9.121907e-001	0.000e+000	9.076e-001	1.655e-001
5	19	-1.009367e+000	0.000e+000	7.326e-001	8.558e-002
6	22	-1.030423e+000	0.000e+000	2.172e-001	6.670e-002
7	25	-1.031578e+000	0.000e+000	4.278e-002	1.444e-002
8	28	-1.031628e+000	0.000e+000	8.777e-003	2.306e-003
9	31	-1.031628e+000	0.000e+000	8.845e-005	2.750e-004
10	34	-1.031628e+000	0.000e+000	8.744e-007	1.354e-006

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>
```

				First-order	Norm of
Iter	F-count	f(x)	Feasibility	optimality	step
0	3	-5.372419e-001	0.000e+000	1.913e+000	

```
... MANY ITERATIONS DELETED ...
```

				First-order	Norm of
9	39	-2.154638e-001	0.000e+000	2.425e-007	6.002e-008

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints were satisfied to within the selected value of the constraint tolerance.

<stopping criteria details>

GlobalSearch stopped because it analyzed all the start points.

All 7 local solver runs converged with a positive local solver exit flag.
```

Setting `GlobalSearch` iterative display, as well as `fmincon` iterative display, yields both displays intermingled.

For an example of iterative display in a parallel environment, see "Example: Parallel MultiStart" on page 3-74.

Global Output Structures

`run` can produce two types of output structures:

- A global output structure. This structure contains information about the overall run from multiple starting points. Details follow.

- Local solver output structures. The vector of `GlobalOptimSolution` objects contains one such structure in each element of the vector. For a description of this structure, see "Output Structures" in the Optimization Toolbox documentation, or the function reference pages for the local solvers: `fmincon`, `fminunc`, `lsqcurvefit`, or `lsqnonlin`.

Global Output Structure

Field	Meaning
funcCount	Total number of calls to user-supplied functions (objective or nonlinear constraint)
localSolverTotal	Number of local solver runs started
localSolverSuccess	Number of local solver runs that finished with a positive exit flag
localSolverIncomplete	Number of local solver runs that finished with a `0` exit flag
localSolverNoSolution	Number of local solver runs that finished with a negative exit flag
message	`GlobalSearch` or `MultiStart` exit message

A positive exit flag from a local solver generally indicates a successful run. A negative exit flag indicates a failure. A `0` exit flag indicates that the solver stopped by exceeding the iteration or function evaluation limit. For more information, see "Exit Flags and Exit Messages" or "Tolerances and Stopping Criteria" in the Optimization Toolbox documentation.

Example: Visualizing the Basins of Attraction

Which start points lead to which basin? For a steepest descent solver, nearby points generally lead to the same basin; see "Basins of Attraction" on page 1-13. However, for Optimization Toolbox solvers, basins are more complicated.

Plot the MultiStart start points from the example, "Example of Run with MultiStart" on page 3-20, color-coded with the basin where they end.

```
% s = RandStream('mt19937ar','Seed',14);
% RandStream.setDefaultStream(s);
% Uncomment the previous lines to get the same output
ms = MultiStart;
opts = optimset('Algorithm','interior-point');
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
+ x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
'objective',sixmin,'lb',[-3,-3],'ub',[3,3],...
'options',opts);
[xminm,fminm,flagm,outptm,manyminsm] = run(ms,problem,50);

possColors = 'kbgcrm';
hold on
for i = 1:size(manyminsm,2)

    % Color of this line
    cIdx = rem(i-1, length(possColors)) + 1;
    color = possColors(cIdx);

    % Plot start points
    u = manyminsm(i).X0;
    x0ThisMin = reshape([u{:}], 2, length(u));
    plot(x0ThisMin(1, :), x0ThisMin(2, :), '.', ...
        'Color',color,'MarkerSize',25);

    % Plot the basin with color i
    plot(manyminsm(i).X(1), manyminsm(i).X(2), '*', ...
        'Color', color, 'MarkerSize',25);
end % basin center marked with a *, start points with dots
hold off
```
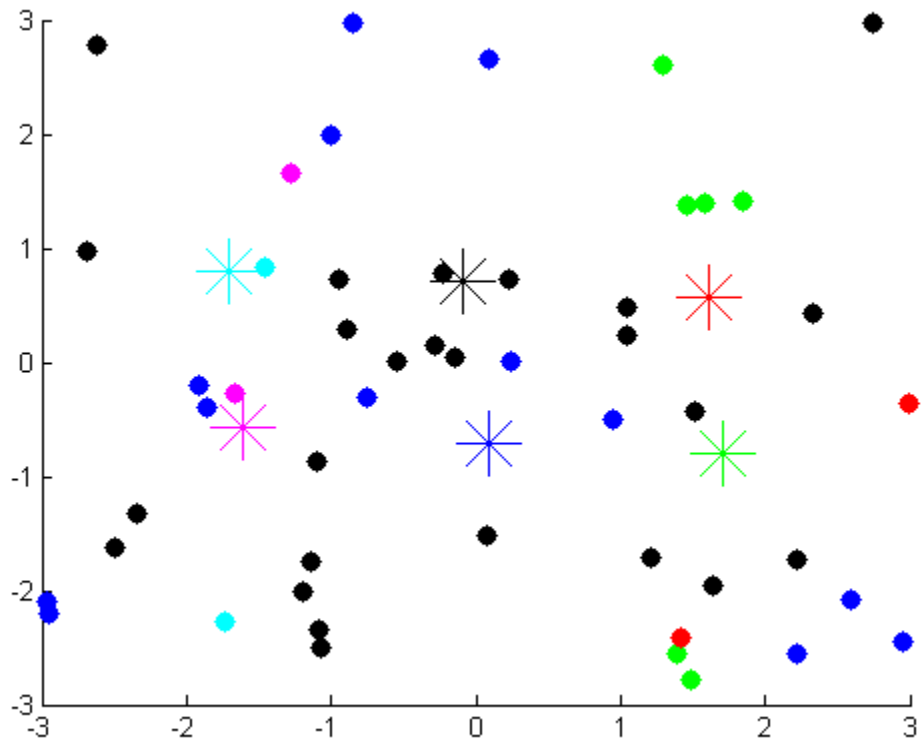
The figure shows the centers of the basins by colored * symbols. Start points with the same color as the * symbol converge to the center of the * symbol.

Start points do not always converge to the closest basin. For example, the red points are closer to the green basin center than to the red basin center. Also, many black and blue start points are closer to the opposite basin centers.

The magenta and red basins are shallow, as you can see in the following contour plot.

sixmin([x,y])
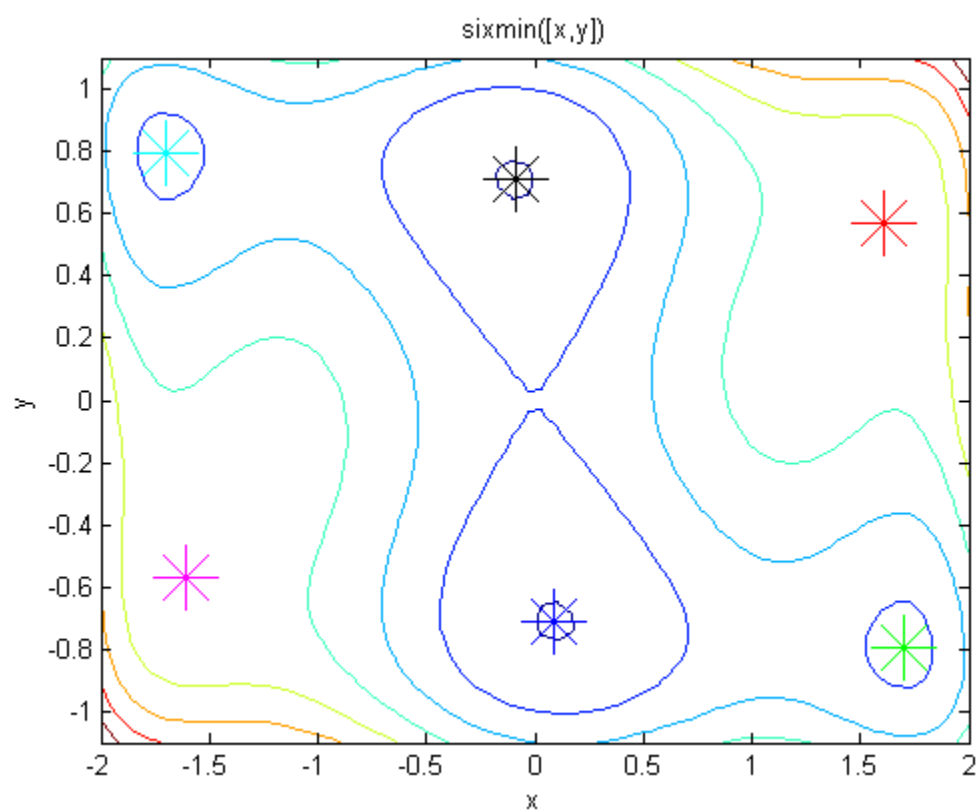
How GlobalSearch and MultiStart Work

In this section...

Multiple Runs of a Local Solver

GlobalSearch and MultiStart have similar approaches to finding global or multiple minima. Both algorithms start a local solver (such as fmincon) from multiple start points. The algorithms use multiple start points to sample multiple basins of attraction. For more information, see "Basins of Attraction" on page 1-13.

Differences Between the Solver Objects

GlobalSearch and MultiStart Algorithm Overview on page 3-36 contains a sketch of the GlobalSearch and MultiStart algorithms.

GlobalSearch Algorithm

```
┌────────────────────────────┐
│ Run fmincon from x0        │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Generate trial points      │
│ (potential start points)   │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Stage 1:                   │
│ Run best start point among the first │
│ NumStageOnePoints trial points │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Stage 2:                   │
│ Loop through remaining trial points, │
│ run fmincon if point satisfies │
│ basin, score, and constraint filters │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Create GlobalOptimSolutions vector │
└────────────────────────────┘
```

MultiStart Algorithm

```
┌────────────────────────────┐
│ Generate start points      │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Run start points           │
└────────────────────────────┘
              │
              ▼
┌────────────────────────────┐
│ Create GlobalOptimSolutions vector │
└────────────────────────────┘
```
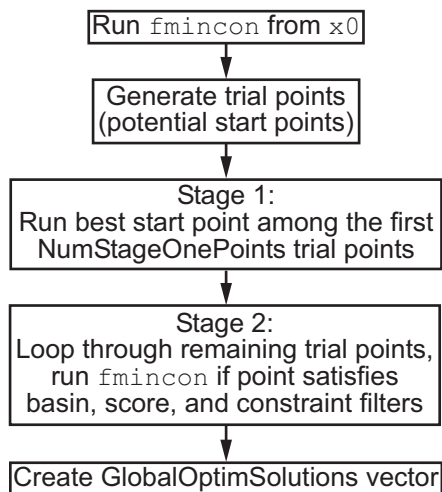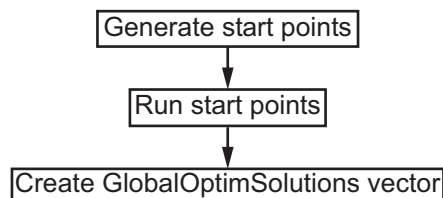
GlobalSearch and MultiStart Algorithm Overview

The main differences between GlobalSearch and MultiStart are:

- GlobalSearch uses a scatter-search mechanism for generating start points. MultiStart uses uniformly distributed start points within bounds, or user-supplied start points.

- GlobalSearch analyzes start points and rejects those points that are unlikely to improve the best local minimum found so far. MultiStart runs all start points (or, optionally, all start points that are feasible with respect to bounds or inequality constraints).

- MultiStart gives a choice of local solver: fmincon, fminunc, lsqcurvefit, or lsqnonlin. The GlobalSearch algorithm uses fmincon.

- MultiStart can run in parallel, distributing start points to multiple processors for local solution. To run MultiStart in parallel, see "How to Use Parallel Processing" on page 8-11.

Deciding Which Solver to Use

The differences between these solver objects boil down to the following decision on which to use:

- Use GlobalSearch to find a single global minimum most efficiently on a single processor.
- Use MultiStart to:
  - Find multiple local minima.
  - Run in parallel.
  - Use a solver other than fmincon.
  - Search thoroughly for a global minimum.
  - Explore your own start points.

GlobalSearch Algorithm

For a description of the algorithm, see Ugray et al. [1].

When you run a GlobalSearch object, the algorithm performs the following steps:

- "Run fmincon from x0" on page 3-38
- "Generate Trial Points" on page 3-38
- "Obtain Stage 1 Start Point, Run" on page 3-38
- "Initialize Basins, Counters, Threshold" on page 3-38
- "Begin Main Loop" on page 3-39
- "Examine Stage 2 Trial Point to See if fmincon Runs" on page 3-39
- "When fmincon Runs:" on page 3-39
- "When fmincon Does Not Run:" on page 3-40
- "Create GlobalOptimSolution" on page 3-41

Run fmincon from x0

GlobalSearch runs fmincon from the start point you give in the problem structure. If this run converges, GlobalSearch records the start point and end point for an initial estimate on the radius of a basin of attraction. Furthermore, GlobalSearch records the final objective function value for use in the *score* function (see "Obtain Stage 1 Start Point, Run" on page 3-38).

The score function is the sum of the objective function value at a point and a multiple of the sum of the constraint violations. So a feasible point has score equal to its objective function value. The multiple for constraint violations is initially 1000. GlobalSearch updates the multiple during the run.

Generate Trial Points

GlobalSearch uses the scatter search algorithm to generate a set of NumTrialPoints trial points. Trial points are potential start points. For a description of the scatter search algorithm, see Glover [2]. The trial points have components in the range (–1e4+1,1e4+1). This range is not symmetric about the origin so that the origin is not in the scatter search.

Obtain Stage 1 Start Point, Run

GlobalSearch evaluates the score function of a set of NumStageOnePoints trial points. It then takes the point with the best score and runs fmincon from that point. GlobalSearch removes the set of NumStageOnePoints trial points from its list of points to examine.

Initialize Basins, Counters, Threshold

The localSolverThreshold is initially the smaller of the two objective function values at the solution points. The solution points are the fmincon solutions starting from x0 and from the Stage 1 start point. If both of these solution points do not exist or are infeasible, localSolverThreshold is initially the penalty function value of the Stage 1 start point.

The GlobalSearch heuristic assumption is that basins of attraction are spherical. The initial estimate of basins of attraction for the solution point from x0 and the solution point from Stage 1 are spheres centered at the solution points. The radius of each sphere is the distance from the initial point to the solution point. These estimated basins can overlap.

There are two sets of counters associated with the algorithm. Each counter is the number of consecutive trial points that:

- Lie within a basin of attraction. There is one counter for each basin.

- Have score function greater than `localSolverThreshold`. For a definition of the score, see "Run fmincon from x0" on page 3-38.

All counters are initially 0.

Begin Main Loop

`GlobalSearch` repeatedly examines a remaining trial point from the list, and performs the following steps. It continually monitors the time, and stops the search if elapsed time exceeds `MaxTime` seconds.

Examine Stage 2 Trial Point to See if fmincon Runs

Call the trial point `p`. Run `fmincon` from `p` if the following conditions hold:

- `p` is not in any existing basin. The criterion for every basin `i` is:

    `|p - center(i)| > DistanceThresholdFactor * radius(i).`

  `DistanceThresholdFactor` is an option (default value `0.75`).

  `radius` is an estimated radius that updates in Update Basin Radius and Threshold and React to Large Counter Values.

- score(p) < `localSolverThreshold`.

- (optional) `p` satisfies bound and/or inequality constraints. This test occurs if you set the `StartPointsToRun` property of the `GlobalSearch` object to `'bounds'` or `'bounds-ineqs'`.

When fmincon Runs:

1 Reset CountersSet the counters for basins and threshold to 0.

2 Update Solution SetIf `fmincon` runs starting from `p`, it can yield a positive exit flag, which indicates convergence. In that case, `GlobalSearch` updates the vector of `GlobalOptimSolution` objects. Call the solution point `xp` and the objective function value `fp`. There are two cases:

- For every other solution point xq with objective function value fq,

  `|xq - xp| > TolX * max(1,|xp|)`

  or

  `|fq - fp| > TolFun * max(1,|fp|).`

  In this case, GlobalSearch creates a new element in the vector of GlobalOptimSolution objects. For details of the information contained in each object, see GlobalOptimSolution.

- For some other solution point xq with objective function value fq,

  `|xq - xp| <= TolX * max(1,|xp|)`

  and

  `|fq - fp| <= TolFun * max(1,|fp|).`

  In this case, GlobalSearch regards xp as equivalent to xq. The GlobalSearch algorithm modifies the GlobalOptimSolution of xq by adding p to the cell array of X0 points.

  There is one minor tweak that can happen to this update. If the exit flag for xq is greater than 1, and the exit flag for xp is 1, then xp replaces xq. This replacement can lead to some points in the same basin being more than a distance of TolX from xp.

3 Update Basin Radius and ThresholdIf the exit flag of the current fmincon run is positive:

  a Set threshold to the score value at start point p.

  b Set basin radius for xp equal to the maximum of the existing radius (if any) and the distance between p and xp.

4 Report to Iterative DisplayWhen the GlobalSearch Display property is `'iter'`, every point that fmincon runs creates one line in the GlobalSearch iterative display.

When fmincon Does Not Run:

1 Update CountersIncrement the counter for every basin containing p. Reset the counter of every other basin to 0.

Increment the threshold counter if score(p) >= localSolverThreshold. Otherwise, reset the counter to 0.

2 React to Large Counter ValuesFor each basin with counter equal to MaxWaitCycle, multiply the basin radius by 1 − BasinRadiusFactor. Reset the counter to 0. (Both MaxWaitCycle and BasinRadiusFactor are settable properties of the GlobalSearch object.)

If the threshold counter equals MaxWaitCycle, increase the threshold:

new threshold = threshold + PenaltyThresholdFactor*(1 + abs(threshold)).

Reset the counter to 0.

3 Report to Iterative DisplayEvery 200th trial point creates one line in the GlobalSearch iterative display.

Create GlobalOptimSolution

After reaching MaxTime seconds or running out of trial points, GlobalSearch creates a vector of GlobalOptimSolution objects. GlobalSearch orders the vector by objective function value, from lowest (best) to highest (worst). This concludes the algorithm.

MultiStart Algorithm

When you run a MultiStart object, the algorithm performs the following steps:

- "Generate Start Points" on page 3-41
- "Filter Start Points (Optional)" on page 3-42
- "Run Local Solver" on page 3-42
- "Check Stopping Conditions" on page 3-43
- "Create GlobalOptimSolution Object" on page 3-43

Generate Start Points

If you call MultiStart with the syntax

```
[x fval] = run(ms,problem,k)
```

for an integer k, MultiStart generates k - 1 start points exactly as if you used a RandomStartPointSet object. The algorithm also uses the x0 start point from the problem structure, for a total of k start points.

A RandomStartPointSet object does not have any points stored inside the object. Instead, MultiStart calls the list method, which generates random points within the bounds given by the problem structure. If an unbounded component exists, list uses an artificial bound given by the ArtificialBound property of the RandomStartPointSet object.

If you provide a CustomStartPointSet object, MultiStart does not generate start points, but uses the points in the object.

Filter Start Points (Optional)

If you set the StartPointsToRun property of the MultiStart object to 'bounds' or 'bounds-ineqs', MultiStart does not run the local solver from infeasible start points. In this context, "infeasible" means start points that do not satisfy bounds, or start points that do not satisfy both bounds and inequality constraints.

The default setting of StartPointsToRun is 'all'. In this case, MultiStart does not discard infeasible start points.

Run Local Solver

MultiStart runs the local solver specified in problem.solver, starting at the points that pass the StartPointsToRun filter. If MultiStart is running in parallel, it sends start points to worker processors one at a time, and the worker processors run the local solver.

The local solver checks whether MaxTime seconds have elapsed at each of its iterations. If so, it exits that iteration without reporting a solution.

When the local solver stops, MultiStart stores the results and continues to the next step.

Report to Iterative Display. When the `MultiStart Display` property is `'iter'`, every point that the local solver runs creates one line in the `MultiStart` iterative display.

Check Stopping Conditions

`MultiStart` stops when it runs out of start points. It also stops when it exceeds a total run time of `MaxTime` seconds.

Create GlobalOptimSolution Object

After `MultiStart` reaches a stopping condition, the algorithm creates a vector of `GlobalOptimSolution` objects as follows:

1 Sort the local solutions by objective function value (`Fval`) from lowest to highest. For the `lsqnonlin` and `lsqcurvefit` local solvers, the objective function is the norm of the residual.

2 Loop over the local solutions `j` beginning with the lowest (best) `Fval`.

3 Find all the solutions `k` satisfying both:

```
|Fval(k) - Fval(j)| <= TolFun*max(1,|Fval(j)|)

|x(k) - x(j)|  <= TolX*max(1,|x(j)|)
```

4 Record `j`, `Fval(j)`, the local solver `output` structure for `j`, and a cell array of the start points for `j` and all the `k`. Remove those points `k` from the list of local solutions. This point is one entry in the vector of `GlobalOptimSolution` objects.

The resulting vector of `GlobalOptimSolution` objects is in order by `Fval`, from lowest (best) to highest (worst).

Report to Iterative Display. After examining all the local solutions, `MultiStart` gives a summary to the iterative display. This summary includes the number of local solver runs that converged, the number that failed to converge, and the number that had errors.

Bibliography

[1] Ugray, Zsolt, Leon Lasdon, John C. Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization.* INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

[2] Glover, F. "A template for scatter search and path relinking." *Artificial Evolution* (J.-K. Hao, E.Lutton, E.Ronald, M.Schoenauer, D.Snyers, eds.). Lecture Notes in Computer Science, 1363, Springer, Berlin/Heidelberg, 1998, pp. 13–54.

[3] Dixon, L. and G. P. Szegö. "The Global Optimization Problem: an Introduction." *Towards Global Optimisation 2* (Dixon, L. C. W. and G. P. Szegö, eds.). Amsterdam, The Netherlands: North Holland, 1978.

Improving Results

In this section...
"Can You Certify a Solution Is Global?" on page 3-45
"Refining the Start Points" on page 3-48
"Changing Options" on page 3-55
"Reproducing Results" on page 3-59

Can You Certify a Solution Is Global?

How can you tell if you have located the global minimum of your objective function? The short answer is that you cannot; you have no guarantee that the result of a Global Optimization Toolbox solver is a global optimum.

However, you can use the following strategies for investigating solutions:

- "Check if a Solution Is a Local Solution with patternsearch" on page 3-45
- "Identify a Bounded Region That Contains a Global Solution" on page 3-46
- "Use MultiStart with More Start Points" on page 3-47

Check if a Solution Is a Local Solution with patternsearch

Before you can determine if a purported solution is a global minimum, first check that it is a local minimum. To do so, run patternsearch on the problem.

To convert the problem to use patternsearch instead of fmincon or fminunc, enter

```
problem.solver = 'patternsearch';
```

Also, change the start point to the solution you just found:

```
problem.x0 = x;
```

For example, Check Nearby Points (in the Optimization Toolbox documentation) shows the following:

```
options = optimset('Algorithm','active-set');
```

```
ffun = @(x)(x(1)-(x(1)-x(2))^2);
problem = createOptimProblem('fmincon', ...
    'objective',ffun,'x0',[1/2 1/3], ...
    'lb',[0 -1],'ub',[1 1],'options',options);
[x fval exitflag] = fmincon(problem)

x =
  1.0e-007 *
        0    0.1614

fval =
 -2.6059e-016

exitflag =
      1
```

However, checking this purported solution with patternsearch shows that there is a better solution. Start patternsearch from the reported solution x:

```
% set the candidate solution x as the start point
problem.x0 = x;
problem.solver = 'patternsearch';
[xp fvalp exitflagp] = patternsearch(problem)

xp =
    1.0000   -1.0000

fvalp =
   -3.0000

exitflagp =
      1
```

Identify a Bounded Region That Contains a Global Solution

Suppose you have a smooth objective function in a bounded region. Given enough time and start points, MultiStart eventually locates a global solution.

Therefore, if you can bound the region where a global solution can exist, you can obtain some degree of assurance that MultiStart locates the global solution.

For example, consider the function

$$f = x^6 + y^6 + \sin(x+y)\left(x^2 + y^2\right) - \cos\left(\frac{x^2}{1+y^2}\right)\left(2 + x^4 + x^2 y^2 + y^4\right).$$

The initial summands $x^6 + y^6$ force the function to become large and positive for large values of $|x|$ or $|y|$. The components of the global minimum of the function must be within the bounds

$$-10 \le x,y \le 10,$$

since 10^6 is much larger than all the multiples of 10^4 that occur in the other summands of the function.

You can identify smaller bounds for this problem; for example, the global minimum is between –2 and 2. It is more important to identify reasonable bounds than it is to identify the best bounds.

Use MultiStart with More Start Points

To check whether there is a better solution to your problem, run `MultiStart` with additional start points. Use `MultiStart` instead of `GlobalSearch` for this task because `GlobalSearch` does not run the local solver from all start points.

For example, see "Example: Searching for a Better Solution" on page 3-52.

Updating Unconstrained Problem from GlobalSearch. If you use `GlobalSearch` on an unconstrained problem, change your `problem` structure before using `MultiStart`. You have two choices in updating a `problem` structure for an unconstrained problem using `MultiStart`:

• Change the `solver` field to `'fminunc'`:

```
problem.solver = 'fminunc';
```

To avoid a warning if your objective function does not compute a gradient, change the local `options` structure to have `LargeScale` set to `'off'`:

```
problem.options.LargeScale = 'off';
```

- Add an artificial constraint, retaining `fmincon` as the local solver:

  ```
  problem.lb = -Inf;
  ```

To search a larger region than the default, see "Refining the Start Points" on page 3-48.

Refining the Start Points

If some components of your problem are unconstrained, `GlobalSearch` and `MultiStart` use artificial bounds to generate random start points uniformly in each component. However, if your problem has far-flung minima, you need widely dispersed start points to find these minima.

Use these methods to obtain widely dispersed start points:

- Give widely separated bounds in your `problem` structure.

- Use a `RandomStartPointSet` object with the `MultiStart` algorithm. Set a large value of the `ArtificialBound` property in the `RandomStartPointSet` object.

- Use a `CustomStartPointSet` object with the `MultiStart` algorithm. Use widely dispersed start points.

There are advantages and disadvantages of each method.

Method	Advantages	Disadvantages
Give bounds in `problem`	Automatic point generation	Makes a more complex Hessian
	Can use with `GlobalSearch`	Unclear how large to set the bounds
	Easy to do	Changes `problem`
	Bounds can be asymmetric	Only uniform points
Large `ArtificialBound` in `RandomStartPointSet`	Automatic point generation	`MultiStart` only
	Does not change `problem`	Only symmetric, uniform points
	Easy to do	Unclear how large to set `ArtificialBound`

Method	Advantages	Disadvantages
CustomStartPointSet	Customizable	MultiStart only
	Does not change problem	Requires programming for generating points

Methods of Generating Start Points

- "Uniform Grid" on page 3-49
- "Perturbed Grid" on page 3-50
- "Widely Dispersed Points for Unconstrained Components" on page 3-50

Uniform Grid. To generate a uniform grid of start points:

1 Generate multidimensional arrays with ndgrid. Give the lower bound, spacing, and upper bound for each component.

   For example, to generate a set of three-dimensional arrays with

   - First component from –2 through 0, spacing 0.5
   - Second component from 0 through 2, spacing 0.25
   - Third component from –10 through 5, spacing 1

   ```
   [X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
   ```

2 Place the arrays into a single matrix, with each row representing one start point. For example:

   ```
   W = [X(:),Y(:),Z(:)];
   ```

   In this example, W is a 720-by-3 matrix.

3 Put the matrix into a CustomStartPointSet object. For example:

   ```
   custpts = CustomStartPointSet(W);
   ```

Call MultiStart run with the CustomStartPointSet object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x fval flag outpt manymins] = run(ms,problem,custpts);
```

Perturbed Grid. Integer start points can yield less robust solutions than slightly perturbed start points.

To obtain a perturbed set of start points:

1 Generate a matrix of start points as in steps 1–2 of "Uniform Grid" on page 3-49.

2 Perturb the start points by adding a random normal matrix with 0 mean and relatively small variance.

For the example in "Uniform Grid" on page 3-49, after making the W matrix, add a perturbation:

```
[X,Y,Z] = ndgrid(-2:.5:0,0:.25:2,-10:5);
W = [X(:),Y(:),Z(:)];
W = W + 0.01*randn(size(W));
```

3 Put the matrix into a CustomStartPointSet object. For example:

```
custpts = CustomStartPointSet(W);
```

Call MultiStart run with the CustomStartPointSet object as the third input. For example,

```
% Assume problem structure and ms MultiStart object exist
[x fval flag outpt manymins] = run(ms,problem,custpts);
```

Widely Dispersed Points for Unconstrained Components. Some components of your problem can lack upper or lower bounds. For example:

• Although no explicit bounds exist, there are levels that the components cannot attain. For example, if one component represents the weight of a single diamond, there is an implicit upper bound of 1 kg (the Hope Diamond is under 10 g). In such a case, give the implicit bound as an upper bound.

• There truly is no upper bound. For example, the size of a computer file in bytes has no effective upper bound. The largest size can be in gigabytes or terabytes today, but in 10 years, who knows?

For truly unbounded components, you can use the following methods of sampling. To generate approximately $1/n$ points in each region $(\exp(n),\exp(n+1))$, use the following formula. If u is random and uniformly distributed from 0 through 1, then $r = 2u - 1$ is uniformly distributed between -1 and 1. Take

$$y = \text{sgn}(r)\big(\exp(1/|r|) - e\big).$$

y is symmetric and random. For a variable bounded below by lb, take

$$y = \text{lb} + \big(\exp(1/u) - e\big).$$

Similarly, for variable bounded above by ub, take

$$y = \text{ub} - \big(\exp(1/u) - e\big).$$

For example, suppose you have a three-dimensional problem with

- x(1) > 0
- x(2) < 100
- x(3) unconstrained

To make 150 start points satisfying these constraints:

```
u = rand(150,3);
r1 = 1./u(:,1);
r1 = exp(r1) - exp(1);
r2 = 1./u(:,2);
r2 = -exp(r2) + exp(1) + 100;
r3 = 1./(2*u(:,3)-1);
r3 = sign(r3).*(exp(abs(r3)) - exp(1));
custpts = CustomStartPointSet([r1,r2,r3]);
```

The following is a variant of this algorithm. Generate a number between 0 and infinity by the method for lower bounds. Use this number as the radius of a point. Generate the other components of the point by taking random numbers for each component and multiply by the radius. You can normalize the random numbers, before multiplying by the radius, so their norm is 1. For

a worked example of this method, see "MultiStart Without Bounds, Widely Dispersed Start Points" on page 3-81.

Example: Searching for a Better Solution

MultiStart fails to find the global minimum in "Multiple Local Minima Via MultiStart" on page 3-67. There are two simple ways to search for a better solution:

- Use more start points

- Give tighter bounds on the search space

Set up the problem structure and MultiStart object:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimset('LargeScale','off'));
ms = MultiStart;
```

Use More Start Points. Run MultiStart on the problem for 200 start points instead of 50:

```
[x fval eflag output manymins] = run(ms,problem,200)

MultiStart completed some of the runs from the start points.

53 out of 200 local solver runs converged with a positive
local solver exit flag.

x =
  1.0e-005 *
    0.7460   -0.2550

fval =
  7.8236e-010
eflag =
    2

output =
```

```
                      funcCount: 32841
                localSolverTotal: 200
              localSolverSuccess: 53
          localSolverIncomplete: 147
          localSolverNoSolution: 0
                        message: [1x143 char]
manymins =
  1x53 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

This time MultiStart found the global minimum, and found 53 total local minima.

To see the range of local solutions, enter hist([manymins.Fval]).
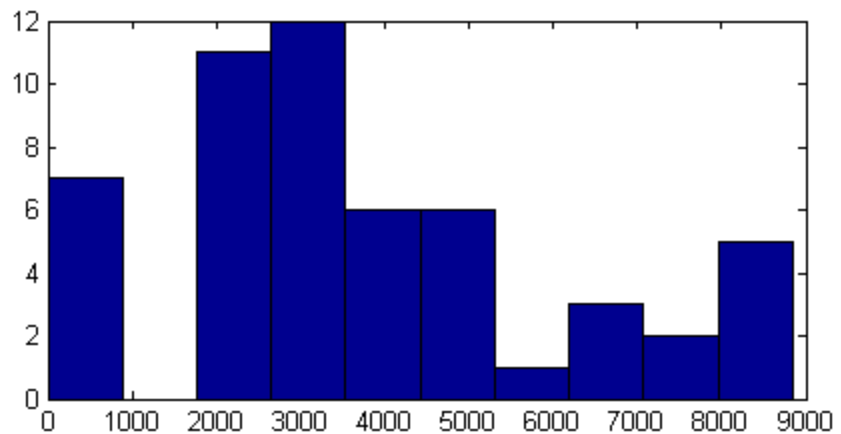
Tighter Bound on the Start Points. Suppose you believe that the interesting local solutions have absolute values of all components less than 100. The default value of the bound on start points is 1000. To use a different value of the bound, generate a RandomStartPointSet with the ArtificialBound property set to 100:

```
startpts = RandomStartPointSet('ArtificialBound',100,...
    'NumStartPoints',50);
[x fval eflag output manymins] = run(ms,problem,startpts)

MultiStart completed some of the runs from the start points.

29 out of 50 local solver runs converged with a
positive local solver exit flag.

x =
  1.0e-007 *
    0.1771    0.3198

fval =
  1.3125e-014

eflag =
     2

output =
                 funcCount: 7557
           localSolverTotal: 50
         localSolverSuccess: 29
      localSolverIncomplete: 21
      localSolverNoSolution: 0
                    message: [1x142 char]

manymins =
  1x25 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
```

```
Output
X0
```

`MultiStart` found the global minimum, and found 25 distinct local solutions. To see the range of local solutions, enter `hist([manymins.Fval])`.
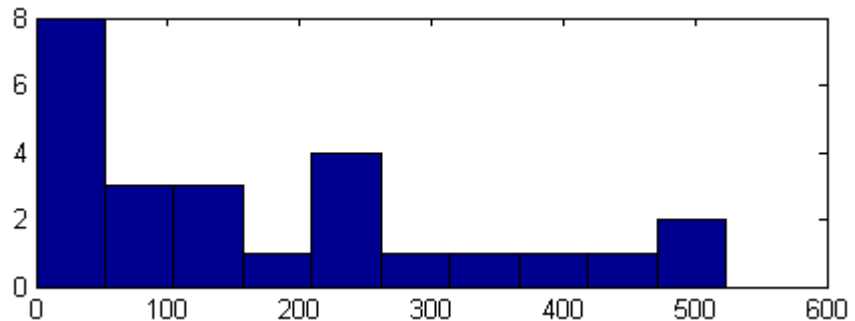
Compared to the minima found in "Use More Start Points" on page 3-52, this run found better (smaller) minima, and had a higher percentage of successful runs.

Changing Options

How to Determine Which Options to Change

After you run a global solver, you might want to change some global or local options. To determine which options to change, the guiding principle is:

- To affect the local solver, set local solver options.

- To affect the start points or solution set, change the `problem` structure, or set the global solver object properties.

For example, to obtain:

- More local minima — Set global solver object properties.

- Faster local solver iterations — Set local solver options.

- Different tolerances for considering local solutions identical (to obtain more or fewer local solutions) — Set global solver object properties.

- Different information displayed at the command line — Decide if you want iterative display from the local solver (set local solver options) or global information (set global solver object properties).

- Different bounds, to examine different regions — Set the bounds in the `problem` structure.

Examples of Choosing Problem Options.

- To start your local solver at points only satisfying inequality constraints, set the `StartPointsToRun` property in the global solver object to `'bounds-ineqs'`. This setting can speed your solution, since local solvers do not have to attempt to find points satisfying these constraints. However, the setting can result in many fewer local solver runs, since the global solver can reject many start points. For an example, see "Example: Using Only Feasible Start Points" on page 3-70.

- To use the `fmincon interior-point` algorithm, set the local solver `Algorithm` option to `'interior-point'`. For an example showing how to do this, see "Examples of Updating Problem Options" on page 3-57.

- For your local solver to have different bounds, set the bounds in the `problem` structure. Examine different regions by setting bounds.

- To see every solution that has positive local exit flag, set the `TolX` property in the global solver object to `0`. For an example showing how to do this, see "Changing Global Options" on page 3-57.

Changing Local Solver Options

There are several ways to change values in a local options structure:

- Update the values using dot addressing and `optimset`. The syntax is

    ```
    problem.options =
    optimset(problem.options,'Parameter',value,...);
    ```

    You can also replace the local options entirely:

    ```
    problem.options = optimset('Parameter',value,...);
    ```

- Use dot addressing on one local option. The syntax is

```
problem.options.Parameter = newvalue;
```

- Recreate the entire problem structure. For details, see "Create a Problem Structure" on page 3-4.

Examples of Updating Problem Options.

1 Create a problem structure:

```
problem = createOptimProblem('fmincon','x0',[-1 2], ...
    'objective',@rosenboth);
```

2 Set the problem to use the sqp algorithm in fmincon:

```
problem.options.Algorithm = 'sqp';
```

3 Update the problem to use the gradient in the objective function, have a TolFun value of 1e-8, and a TolX value of 1e-7:

```
problem.options = optimset(problem.options,'GradObj','on', ...
    'TolFun',1e-8,'TolX',1e-7);
```

Changing Global Options

There are several ways to change characteristics of a GlobalSearch or MultiStart object:

- Use dot addressing. For example, suppose you have a default MultiStart object:

```
ms = MultiStart

  Properties:
        UseParallel: 'never'
    StartPointsToRun: 'all'
            Display: 'final'
             TolFun: 1.0000e-006
               TolX: 1.0000e-006
            MaxTime: Inf
```

To change ms to have its TolX value equal to 1e-3, update the TolX field:

```
ms.TolX = 1e-3

   Properties:
          UseParallel: 'never'
     StartPointsToRun: 'all'
              Display: 'final'
               TolFun: 1.0000e-006
                 TolX: 1.0000e-003
              MaxTime: Inf
```

- Reconstruct the object starting from the current settings. For example, to set the TolFun field in ms to 1e-3, retaining the nondefault value for TolX:

```
ms = MultiStart(ms,'TolFun',1e-3)

   Properties:
          UseParallel: 'never'
     StartPointsToRun: 'all'
              Display: 'final'
               TolFun: 1.0000e-003
                 TolX: 1.0000e-003
              MaxTime: Inf
```

- Convert a GlobalSearch object to a MultiStart object, or vice-versa. For example, with the ms object from the previous example, create a GlobalSearch object with the same values of TolX and TolFun:

```
gs = GlobalSearch(ms)

   Properties:
              NumTrialPoints: 1000
           BasinRadiusFactor: 0.2000
     DistanceThresholdFactor: 0.7500
                MaxWaitCycle: 20
            NumStageOnePoints: 200
       PenaltyThresholdFactor: 0.2000
            StartPointsToRun: 'all'
                     Display: 'final'
                      TolFun: 1.0000e-003 % from ms
                        TolX: 1.0000e-003 % from ms
                     MaxTime: Inf
```

Reproducing Results

GlobalSearch and MultiStart use pseudorandom numbers in choosing start points. Use the same pseudorandom number stream again to:

- Compare various algorithm settings.

- Have an example run repeatably.

- Extend a run, with known initial segment of a previous run.

Both GlobalSearch and MultiStart use the default random number stream.

Steps to Take in Reproducing Results

1 Before running your problem, store the current state of the default random number stream:

```
stream = RandStream.getDefaultStream;
strmstate = stream.State;
```

2 Run your GlobalSearch or MultiStart problem.

3 Restore the state of the random number stream:

```
stream.State = strmstate;
```

4 If you run your problem again, you get the same result.

Example: Reproducing a GlobalSearch or MultiStart Result

This example shows how to obtain reproducible results for "Example: Finding Global or Multiple Local Minima" on page 3-63. The example follows the procedure in "Steps to Take in Reproducing Results" on page 3-59.

1 Store the current state of the default random number stream:

```
stream = RandStream.getDefaultStream;
strmstate = stream.State;
```

2 Create the sawtoothxy function file:

```
function f = sawtoothxy(x,y)
```

```
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

3 Create the problem structure and GlobalSearch object:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimset('Algorithm','sqp'));
gs = GlobalSearch('Display','iter');
```

4 Run the problem:

```
[x fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	465	422.9			422.9	2	Initial Point
200	1730	1.547e-015			1.547e-015	1	Stage 1 Local
300	1830	1.547e-015	6.01e+004	1.074			Stage 2 Search
400	1930	1.547e-015	1.47e+005	4.16			Stage 2 Search
500	2030	1.547e-015	2.63e+004	11.84			Stage 2 Search
600	2130	1.547e-015	1.341e+004	30.95			Stage 2 Search
700	2230	1.547e-015	2.562e+004	65.25			Stage 2 Search
800	2330	1.547e-015	5.217e+004	163.8			Stage 2 Search
900	2430	1.547e-015	7.704e+004	409.2			Stage 2 Search
981	2587	1.547e-015	42.24	516.6	7.573	1	Stage 2 Local
1000	2606	1.547e-015	3.299e+004	42.24			Stage 2 Search

```
GlobalSearch stopped because it analyzed all the trial points.

All 3 local solver runs converged with a positive local solver exit flag.

x =
  1.0e-007 *
    0.0414    0.1298
```

```
fval =
  1.5467e-015
```

You might obtain a different result when running this problem, since the random stream was in an unknown state at the beginning of the run.

5 Restore the state of the random number stream:

```
stream.State = strmstate;
```

6 Run the problem again. You get the same output.

```
[x fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	465	422.9			422.9	2	Initial Point
200	1730	1.547e-015			1.547e-015	1	Stage 1 Local

```
... Output deleted to save space ...

x =
  1.0e-007 *

   0.0414    0.1298


fval =
  1.5467e-015
```

Parallel Processing and Random Number Streams

You obtain reproducible results from MultiStart when you run the algorithm in parallel the same way as you do for serial computation. Runs are reproducible because MultiStart generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random numbers.

To reproduce a parallel `MultiStart` run, use the procedure described in "Steps to Take in Reproducing Results" on page 3-59. For a description of how to run `MultiStart` in parallel, see "How to Use Parallel Processing" on page 8-11.

GlobalSearch and MultiStart Examples

Example: Finding Global or Multiple Local Minima

This example illustrates how GlobalSearch finds a global minimum efficiently, and how MultiStart finds many more local minima.

The objective function for this example has many local minima and a unique global minimum. In polar coordinates, the function is

$$f(r,t) = g(r)h(t),$$

where

$$g(r) = \left(\sin(r) - \frac{\sin(2r)}{2} + \frac{\sin(3r)}{3} - \frac{\sin(4r)}{4} + 4 \right) \frac{r^2}{r+1}$$

$$h(t) = 2 + \cos(t) + \frac{\cos\left(2t - \frac{1}{2}\right)}{2}.$$
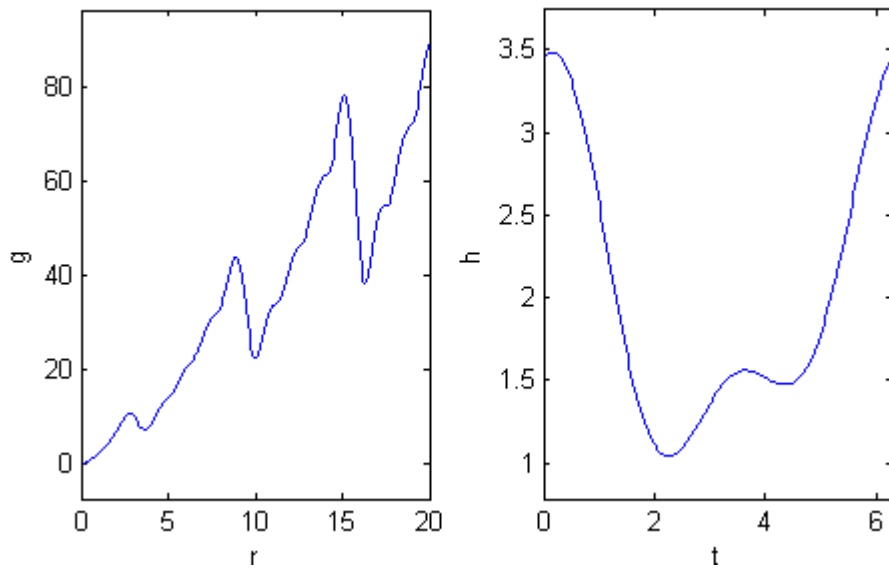
The global minimum is at $r = 0$, with objective function 0. The function $g(r)$ grows approximately linearly in r, with a repeating sawtooth shape. The function $h(t)$ has two local minima, one of which is global.
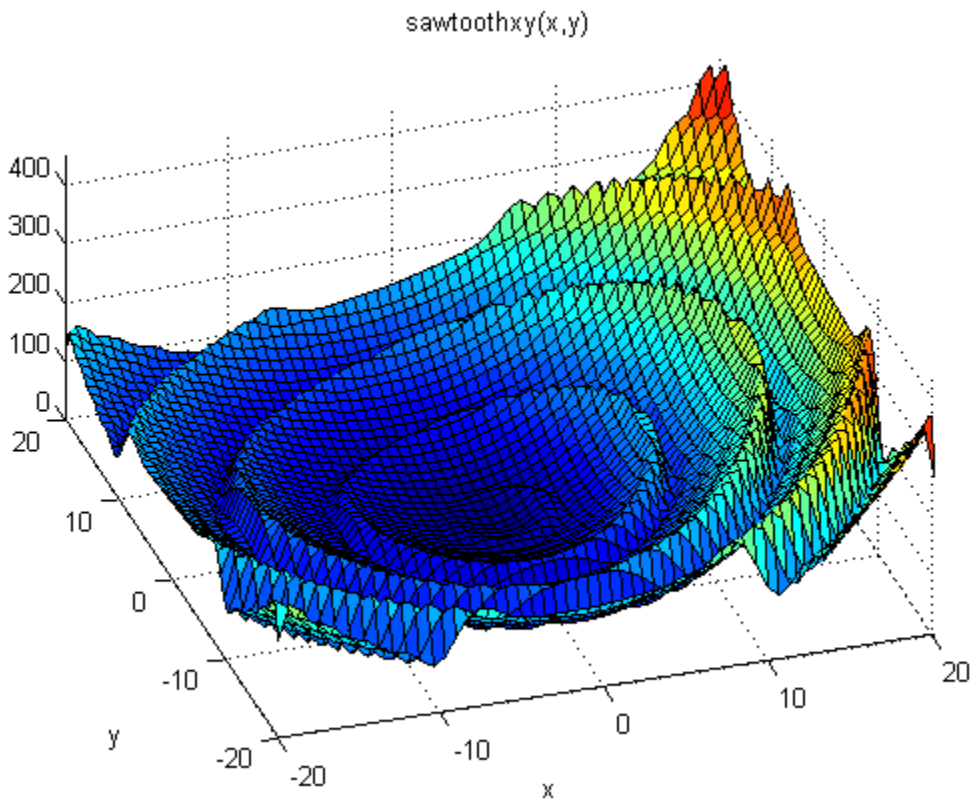
sawtoothxy(x,y)

Single Global Minimum Via GlobalSearch

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure. Use the `'sqp'` algorithm for `fmincon`:

```
problem = createOptimProblem('fmincon',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimset('Algorithm','sqp'));
```

The start point is `[100,-50]` instead of `[0,0]`, so `GlobalSearch` does not start at the global solution.

3 Add an artificial bound, and validate the problem structure by running `fmincon`:

```
problem.lb = -Inf;
[x fval] = fmincon(problem)

x =
   45.6965 -107.6645

fval =
  555.6941
```

4 Create the `GlobalSearch` object, and set iterative display:

```
gs = GlobalSearch('Display','iter');
```

5 Run the solver:

```
[x fval] = run(gs,problem)
```

Num Pts Analyzed	F-count	Best f(x)	Current Penalty	Threshold Penalty	Local f(x)	Local exitflag	Procedure
0	465	422.9			422.9	2	Initial Point
200	1730	1.547e-015			1.547e-015	1	Stage 1 Local
300	1830	1.547e-015	5.709e+004	1.074			Stage 2 Search
400	1930	1.547e-015	1.646e+005	4.16			Stage 2 Search
500	2030	1.547e-015	2.262e+004	11.84			Stage 2 Search
600	2130	1.547e-015	1680	30.95			Stage 2 Search
700	2230	1.547e-015	1.138e+004	65.25			Stage 2 Search
800	2330	1.547e-015	1.573e+005	163.8			Stage 2 Search

```
    900    2430  1.547e-015   3.676e+004      409.2                           Stage 2 Search
    977    3147  1.547e-015         669        707.8        445.5        1     Stage 2 Local
   1000    3170  1.547e-015        1634        505.5                           Stage 2 Search
```

```
GlobalSearch stopped because it analyzed all the trial points.
```

```
All 3 local solver runs converged with a positive local solver exit flag.
```

```
x =
  1.0e-007 *
    0.0414    0.1298
```

```
fval =
  1.5467e-015
```

You can get different results, since GlobalSearch is stochastic.

The solver found three local minima, and it found the global minimum near [0,0].

Multiple Local Minima Via MultiStart

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure. Use the fminunc algorithm with the LargeScale option set to 'off'. The reasons for these choices are:

- The problem is unconstrained. Therefore, fminunc is the appropriate solver; see "Optimization Decision Table" in the Optimization Toolbox documentation.

- The fminunc LargeScale algorithm requires a gradient; see "Choosing the Algorithm" in the Optimization Toolbox documentation. Therefore, set LargeScale to 'off'.

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimset('LargeScale','off'));
```

3 Validate the problem structure by running it:

```
[x fval] = fminunc(problem)

x =
    8.4420 -110.2602

fval =
  435.2573
```

4 Create a default MultiStart object:

```
ms = MultiStart;
```

5 Run the solver for 50 iterations, recording the local minima:

```
[x fval eflag output manymins] = run(ms,problem,50)

MultiStart completed some of the runs from the start points.

16 out of 50 local solver runs converged with a positive
local solver exit flag.

x =
 -379.3434   559.6154

fval =
  1.7590e+003

eflag =
      2
```

```
output =
                    funcCount: 7803
             localSolverTotal: 50
           localSolverSuccess: 16
        localSolverIncomplete: 34
        localSolverNoSolution: 0
                      message: [1x142 char]
manymins =
  1x16 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

You can get different results, since MultiStart is stochastic.

The solver did not find the global minimum near [0,0]. It found 16 distinct local minima.

6 Plot the function values at the local minima:

```
hist([manymins.Fval])
```
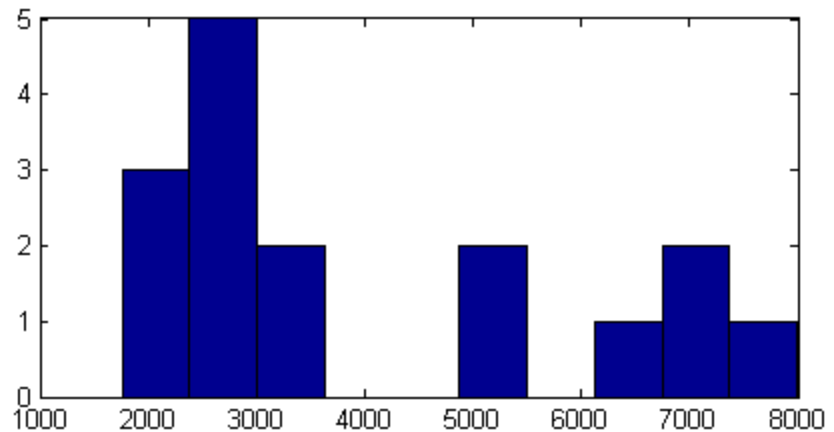
Plot the function values at the three best points:

```
bestf = [manymins.Fval];
hist(bestf(1:3))
```
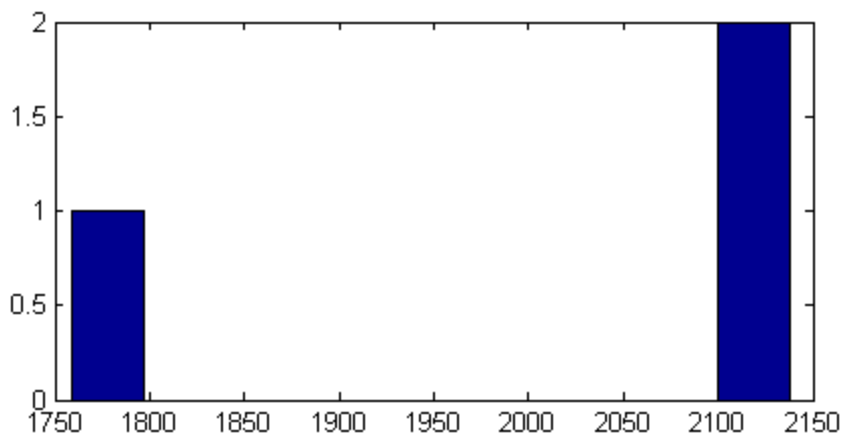

MultiStart started fminunc from start points with components uniformly distributed between –1000 and 1000. fminunc often got stuck in one of the many local minima. fminunc exceeded its iteration limit or function evaluation limit 34 times.

Example: Using Only Feasible Start Points

You can set the StartPointsToRun option so that MultiStart and GlobalSearch use only start points that satisfy inequality constraints. This option can speed your optimization, since the local solver does not have to search for a feasible region. However, the option can cause the solvers to miss some basins of attraction.

There are three settings for the StartPointsToRun option:

- all — Accepts all start points

- bounds — Rejects start points that do not satisfy bounds

- bounds-ineqs — Rejects start points that do not satisfy bounds or inequality constraints

For example, suppose your objective function is

```
function y = tiltcircle(x)
vx = x(:)-[4;4]; % ensure vx is in column form
y = vx'*[1;1] + sqrt(16 - vx'*vx); % complex if norm(x-[4;4])>4
```

`tiltcircle` returns complex values for `norm(x - [4 4]) > 4`.
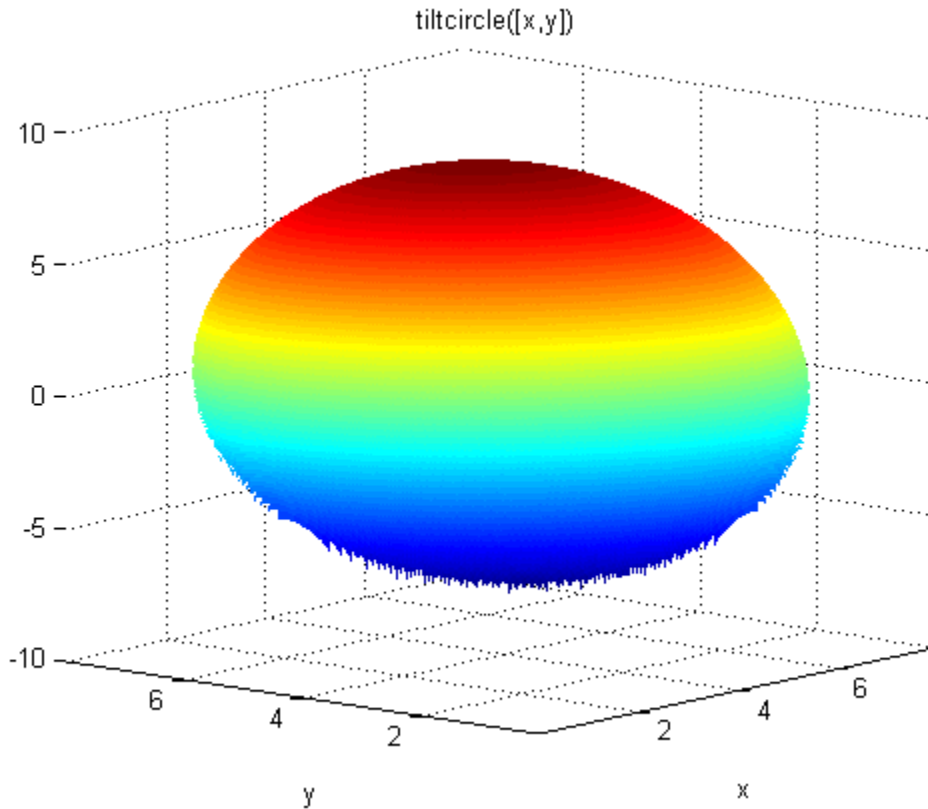
tiltcircle([x,y])

Write a constraint function that is positive on the set where `norm(x - [4 4]) > 4`

```
function [c ceq] = myconstraint(x)
ceq = [];
```

```
cx = x(:) - [4;4]; % ensure x is a column vector
c = cx'*cx - 16; % negative where tiltcircle(x) is real
```

Set GlobalSearch to use only start points satisfying inequality constraints:

```
gs = GlobalSearch('StartPointsToRun','bounds-ineqs');
```

To complete the example, create a problem structure and run the solver:

```
opts = optimset('Algorithm','interior-point');
problem = createOptimProblem('fmincon',...
    'x0',[4 4],'objective',@tiltcircle,...
    'nonlcon',@myconstraint,'lb',[-10 -10],...
    'ub',[10 10],'options',opts);
[x,fval,exitflag,output,solutionset] = run(gs,problem)

GlobalSearch stopped because it analyzed all the trial points.

All 5 local solver runs converged with a positive
local solver exit flag.

x =
    1.1716    1.1716

fval =
   -5.6530

exitflag =
     1

output =
                      funcCount: 3302
                localSolverTotal: 5
              localSolverSuccess: 5
           localSolverIncomplete: 0
           localSolverNoSolution: 0
                         message: [1x137 char]
solutionset =
  1x4 GlobalOptimSolution

  Properties:
```

```
X
Fval
Exitflag
Output
X0
```
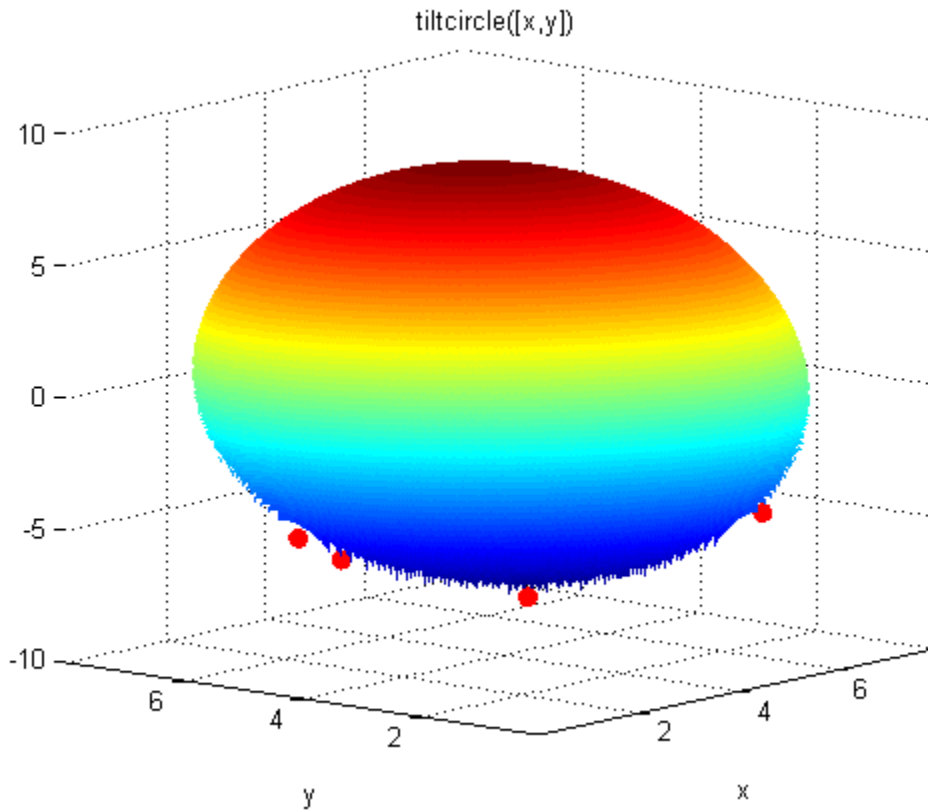

tiltcircle With Local Minima

Why Does GlobalSearch Find Several Local Minima?
The `tiltcircle` function has just one local minimum. Yet GlobalSearch (`fmincon`) stops at several points. Does this mean `fmincon` makes an error?

The reason that `fmincon` stops at several boundary points is subtle. The `tiltcircle` function has an infinite gradient on the boundary, as you can see from a one-dimensional calculation:

$$\frac{d}{dx}\sqrt{16 - x^2} = \frac{-x}{\sqrt{16 - x^2}} = \pm\infty \text{ at } |x| = 4.$$

So there is a huge gradient normal to the boundary. This gradient overwhelms the small additional tilt from the linear term. As far as `fmincon` can tell, boundary points are stationary points for the constrained problem.

This behavior can arise whenever you have a function that has a square root.

Example: Parallel MultiStart

If you have a multicore processor or access to a processor network, you can use Parallel Computing Toolbox™ functions with `MultiStart`. This example shows how to find multiple minima in parallel for a problem, using a processor with two cores. The problem is the same as in "Multiple Local Minima Via MultiStart" on page 3-67.

1 Write a function file to compute the objective:

```
function f = sawtoothxy(x,y)
[t r] = cart2pol(x,y); % change to polar coordinates
h = cos(2*t - 1/2)/2 + cos(t) + 2;
g = (sin(r) - sin(2*r)/2 + sin(3*r)/3 - sin(4*r)/4 + 4) ...
    .*r.^2./(r+1);
f = g.*h;
end
```

2 Create the problem structure:

```
problem = createOptimProblem('fminunc',...
    'objective',@(x)sawtoothxy(x(1),x(2)),...
    'x0',[100,-50],'options',...
    optimset('LargeScale','off'));
```

3 Validate the problem structure by running it:

```
[x fval] = fminunc(problem)

x =
    8.4420 -110.2602

fval =
  435.2573
```

4 Create a `MultiStart` object, and set the object to use parallel processing and iterative display:

```
ms = MultiStart('UseParallel','always','Display','iter');
```

5 Set up parallel processing:

```
matlabpool open 2

Starting matlabpool using the 'local' configuration
... connected to 2 labs.
```

6 Run the problem on 50 start points:

```
[x fval eflag output manymins] = run(ms,problem,50);
Running the local solvers in parallel.

 Run        Local      Local      Local      Local   First-order
Index     exitflag     f(x)      # iter    F-count    optimality
  17          2         3953        4         21        0.1626
  16          0         1331       45        201        65.02
  34          0         7271       54        201        520.9
  33          2         8249        4         18        2.968
    ... Many iterations omitted ...
  47          2         2740        5         21        0.0422
  35          0         8501       48        201        424.8
  50          0         1225       40        201        21.89

MultiStart completed some of the runs from the start points.

17 out of 50 local solver runs converged with a positive
local solver exit flag.
```

Notice that the run indexes look random. Parallel MultiStart runs its start points in an unpredictable order.

Notice that MultiStart confirms parallel processing in the first line of output, which states: "Running the local solvers in parallel."

7 When finished, shut down the parallel environment:

```
matlabpool close
Sending a stop signal to all the labs ... stopped.
```

For an example of how to obtain better solutions to this problem, see "Example: Searching for a Better Solution" on page 3-52. You can use parallel processing along with the techniques described in that example.

Speedup with Parallel Computing

The results of MultiStart runs are stochastic. The timing of runs is stochastic, too. Nevertheless, some clear trends are apparent in the following table. The data for the table came from one run at each number of start points, on a machine with two cores.

Start Points	Parallel Seconds	Serial Seconds
50	3.6	3.4
100	4.9	5.7
200	8.3	10
500	16	23
1000	31	46

Parallel computing can be slower than serial when you use only a few start points. As the number of start points increases, parallel computing becomes increasingly more efficient than serial.

There are many factors that affect speedup (or slowdown) with parallel processing. For more information, see "Improving Performance with Parallel Computing" in the Optimization Toolbox documentation.

Example: Isolated Global Minimum

Finding a start point in the basin of attraction of the global minimum can be difficult when the basin is small or when you are unsure of the location of the minimum. To solve this type of problem you can:

- Add sensible bounds

- Take a huge number of random start points

- Make a methodical grid of start points

- For an unconstrained problem, take widely dispersed random start points

This example shows these methods and some variants.

The function $-\mathrm{sech}(x)$ is nearly 0 for all $|x| > 5$, and $-\mathrm{sech}(0) = -1$. The example is a two-dimensional version of the sech function, with one minimum at [1,1], the other at [1e5,-1e5]:

$$f(x,y) = -10\,\mathrm{sech}(|x - (1,1)|) - 20\,\mathrm{sech}(.0003(|x - (1e5,-1e5)|)) - 1.$$

f has a global minimum of -21 at (1e5,–1e5), and a local minimum of -11 at (1,1).
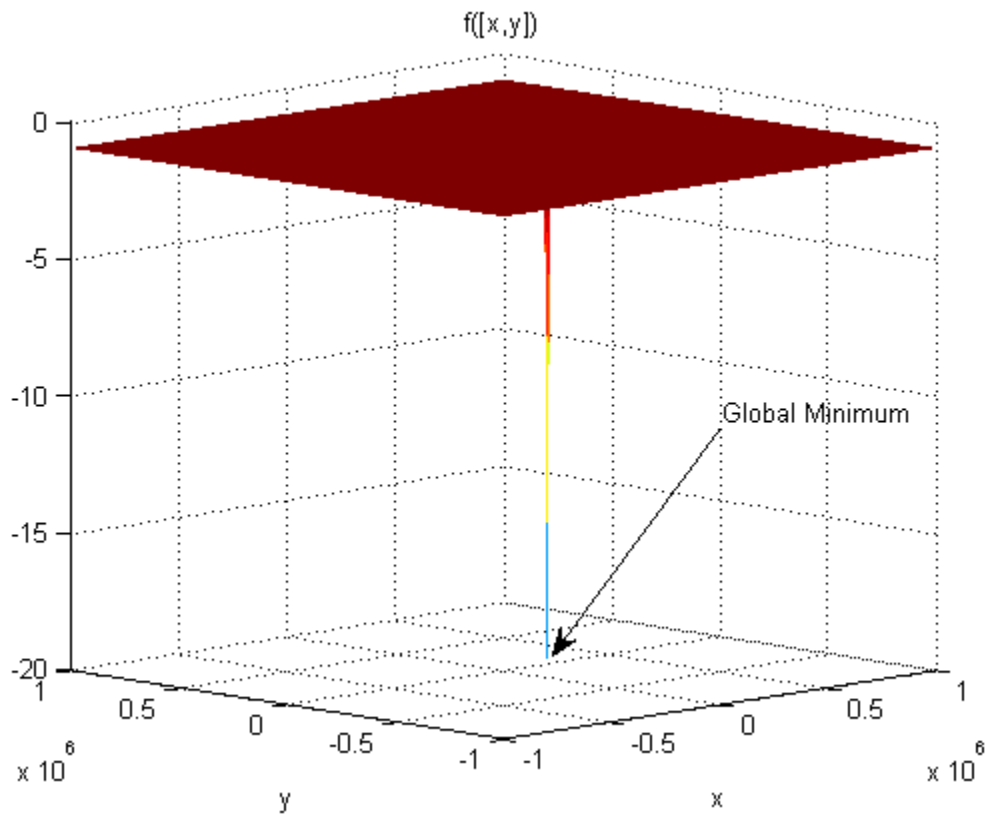
The minimum at (1e5,–1e5) shows as a narrow spike. The minimum at (1,1) does not show since it is too narrow.

The following sections show various methods of searching for the global minimum. Some of the methods are not successful on this problem. Nevertheless, you might find each method useful for different problems.

- "Default Settings Cannot Find the Global Minimum — Add Bounds" on page 3-79

- "GlobalSearch with Bounds and More Start Points" on page 3-79

- "MultiStart with Bounds and Many Start Points" on page 3-80

Default Settings Cannot Find the Global Minimum — Add Bounds

GlobalSearch and MultiStart cannot find the global minimum using default global options, since the default start point components are in the range (–9999,10001) for GlobalSearch and (–1000,1000) for MultiStart.

With additional bounds of –1e6 and 1e6 in problem, GlobalSearch usually does not find the global minimum:

```
x1 = [1;1];x2 = [1e5;-1e5];
f = @(x)-10*sech(norm(x(:)-x1)) -20*sech((norm(x(:)-x2))*3e-4) -1;
problem = createOptimProblem('fmincon','x0',[0,0],'objective',f,...
    'lb',[-1e6;-1e6],'ub',[1e6;1e6]);
gs = GlobalSearch;
[xfinal fval] = run(gs,problem)

GlobalSearch stopped because it analyzed all the trial points.

All 57 local solver runs converged with a positive
local solver exit flag.

xfinal =
    1.0000
    1.0000

fval =
  -11.0000
```

GlobalSearch with Bounds and More Start Points

To find the global minimum, you can search more points. This example uses 1e5 start points, and a MaxTime of 300 s:

```
gs.NumTrialPoints = 1e5;
gs.MaxTime = 300;
```

```
[xg fvalg] = run(gs,problem)

GlobalSearch stopped because maximum time is exceeded.

GlobalSearch called the local solver 2186 times before exceeding
the clock time limit (MaxTime = 300 seconds).
1943 local solver runs converged with a positive
local solver exit flag.

xg =
  1.0e+005 *

    1.0000
   -1.0000

fvalg =
  -21.0000
```

In this case, GlobalSearch found the global minimum.

MultiStart with Bounds and Many Start Points

Alternatively, you can search using MultiStart with many start points. This example uses 1e5 start points, and a MaxTime of 300 s:

```
ms = MultiStart(gs);
[xm fvalm] = run(ms,problem,1e5)

MultiStart stopped because maximum time was exceeded.

MultiStart called the local solver 17266 times before exceeding
the clock time limit (MaxTime = 300 seconds).
17266 local solver runs converged with a positive
local solver exit flag.

xm =
    1.0000
    1.0000

fvalm =
```

```
    -11.0000
```

In this case, `MultiStart` failed to find the global minimum.

MultiStart Without Bounds, Widely Dispersed Start Points

You can also use `MultiStart` to search an unbounded region to find the global minimum. Again, you need many start points to have a good chance of finding the global minimum.

The first five lines of code generate 10,000 widely dispersed random start points using the method described in "Widely Dispersed Points for Unconstrained Components" on page 3-50. `newprob` is a problem structure using the `fminunc` local solver and no bounds:

```
u = rand(1e4,1);
u = 1./u;
u = exp(u) - exp(1);
s = rand(1e4,1)*2*pi;
stpts = [u.*cos(s),u.*sin(s)];
startpts = CustomStartPointSet(stpts);
newprob = createOptimProblem('fminunc','x0',[0;0],'objective',f);
[xcust fcust] = run(ms,newprob,startpts)

MultiStart completed the runs from all start points.

All 10000 local solver runs converged with a positive
local solver exit flag.

xm =
  1.0000e+005

fvalm =
  -21.0000
```

In this case, `MultiStart` found the global minimum.

MultiStart with a Regular Grid of Start Points

You can also use a grid of start points instead of random start points. To learn how to construct a regular grid for more dimensions, or one that has

small perturbations, see "Uniform Grid" on page 3-49 or "Perturbed Grid" on page 3-50.

```
xx = -1e6:1e4:1e6;
[xxx yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
bigstart = CustomStartPointSet(z);
[xgrid fgrid] = run(ms,newprob,bigstart)

MultiStart completed the runs from all start points.

All 10000 local solver runs converged with a positive
local solver exit flag.

xcust =
  1.0e+004 *

   10.0000
  -10.0000

fcust =
  -21.0000
```

In this case, MultiStart found the global minimum.

MultiStart with Regular Grid and Promising Start Points

Making a regular grid of start points, especially in high dimensions, can use an inordinate amount of memory or time. You can filter the start points to run only those with small objective function value.

To perform this filtering most efficiently, write your objective function in a vectorized fashion. For information, see "Example: Writing a Vectorized Function" on page 2-3 or "Vectorizing the Objective and Constraint Functions" on page 4-72. The following function handle computes a vector of objectives based on an input matrix whose rows represent start points:

```
x1 = [1;1];x2 = [1e5;-1e5];
g = @(x) -10*sech(sqrt((x(:,1)-x1(1)).^2 + (x(:,2)-x1(2)).^2)) ...
      -20*sech(sqrt((x(:,1)-x2(1)).^2 + (x(:,2)-x2(2)).^2))-1;
```

Suppose you want to run the local solver only for points where the value is less than –2. Start with a denser grid than in "MultiStart with a Regular Grid of Start Points" on page 3-81, then filter out all the points with high function value:

```
xx = -1e6:1e3:1e6;
[xxx yyy] = meshgrid(xx,xx);
z = [xxx(:),yyy(:)];
idx = g(z) < -2; % index of promising start points
zz = z(idx,:);
smallstartset = CustomStartPointSet(zz);
newprobg = createOptimProblem('fminunc','x0',[0,0],...
    'objective',g); % row vector x0 since g expects rows
[xfew ffew] = run(ms,newprobg,smallstartset)

MultiStart completed the runs from all start points.

All 2 local solver runs converged with a positive
local solver exit flag.

xfew =
      100000     -100000

ffew =
    -21
```

In this case, MultiStart found the global minimum. There are only two start points in smallstartset, one of which is the global minimum.

4

Using 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.

Global Optimization Toolbox functions include three direct search algorithms called the generalized pattern search (GPS) algorithm, the generating set search (GSS) algorithm, and the mesh adaptive search (MADS) algorithm. All 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 GPS algorithm uses fixed direction vectors. The GSS algorithm is identical to the GPS algorithm, except when there are linear constraints, and when the current point is near a linear constraint boundary. The MADS algorithm uses a random selection of vectors to define the mesh. For details, see "Patterns" on page 4-11.

Performing a Pattern Search

In this section...
"Calling patternsearch at the Command Line" on page 4-3
"Using the Optimization Tool for Pattern Search" on page 4-3

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 4-36 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 start 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 a 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 4-8 gives an example of using the Optimization Tool.

The "Optimization Tool" chapter in the Optimization Toolbox documentation 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 Global Optimization Toolbox software. View the code for the function by entering

```
type ps_example
```

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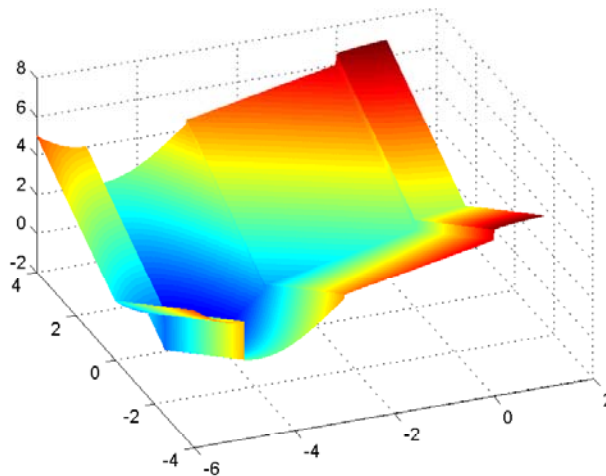
Finding the Minimum of the Function

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].
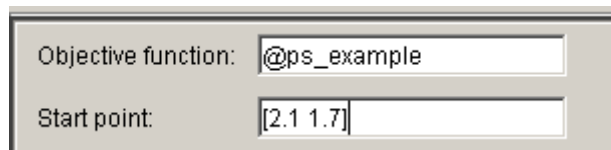
Leave the fields in the **Constraints** pane blank because the problem is unconstrained.

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

The **Run solver and view results** pane displays the results of the pattern search.
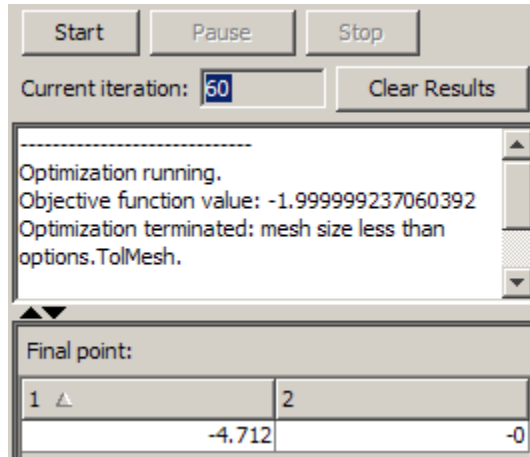
The reason the optimization terminated is that the mesh size became smaller than the acceptable tolerance value for the mesh size, defined by the **Mesh tolerance** parameter in the **Stopping criteria** pane. The minimum function value is approximately –2. The **Final point** pane displays the point 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**
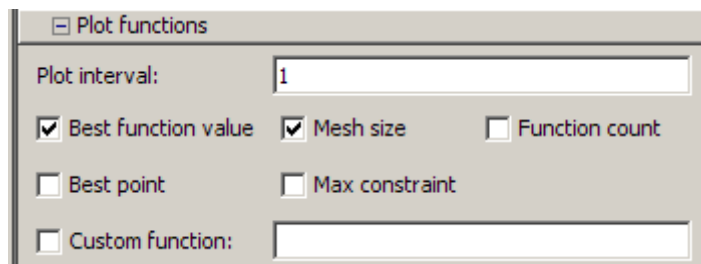
Then click **Start** to run the pattern search. This displays the following plots.
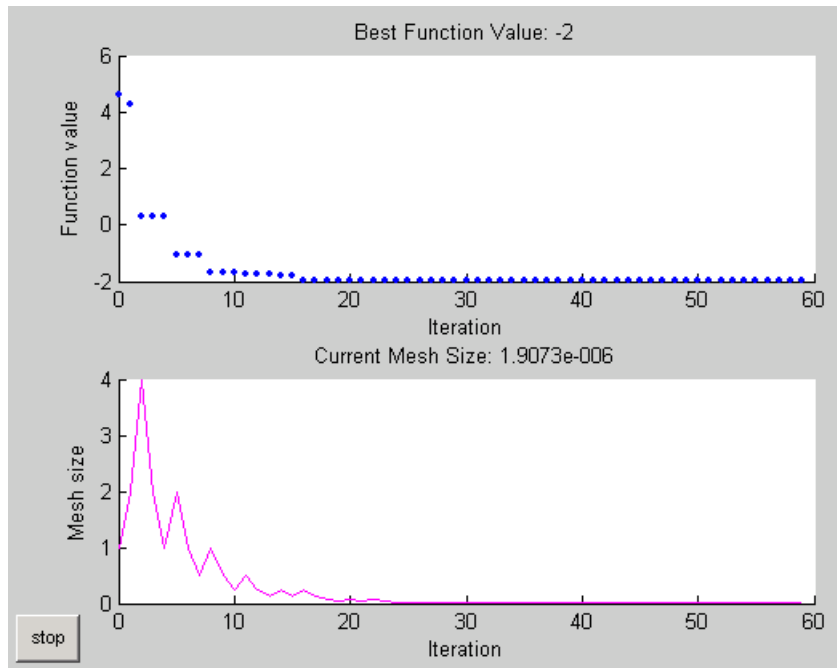
The upper plot shows the objective function value of the best point at each iteration. Typically, the objective function values improve rapidly at the early iterations and then level off as they approach the optimal value.

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 4-14.

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 2*N* 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 2*N* 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 GSS, the pattern is identical to the GPS pattern, except when there are linear constraints and the current point is near a constraint boundary. For a description of the way in which GSS forms a pattern with linear constraints, see Kolda, Lewis, and Torczon [1]. The GSS algorithm is more efficient than the GPS algorithm when you have linear constraints. For an example showing the efficiency gain, see "Example: Comparing the Efficiency of Poll Options" on page 4-48.

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 2*N* or *N*+1. As in GPS, 2*N* 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.

[1] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

Meshes

At each step, `patternsearch` searches a set of points, called a *mesh*, for a point that improves the objective function. `patternsearch` forms 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

In this section...
"Context" on page 4-14
"Successful Polls" on page 4-15
"An Unsuccessful Poll" on page 4-18
"Displaying the Results at Each Iteration" on page 4-19
"More Iterations" on page 4-20
"Stopping Conditions for the Pattern Search" on page 4-21
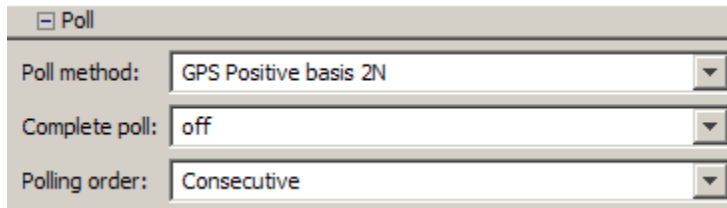
Context

patternsearch finds a sequence of points, x0, x1, x2, ... , that approach 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 4-7.

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.

The problem setup:

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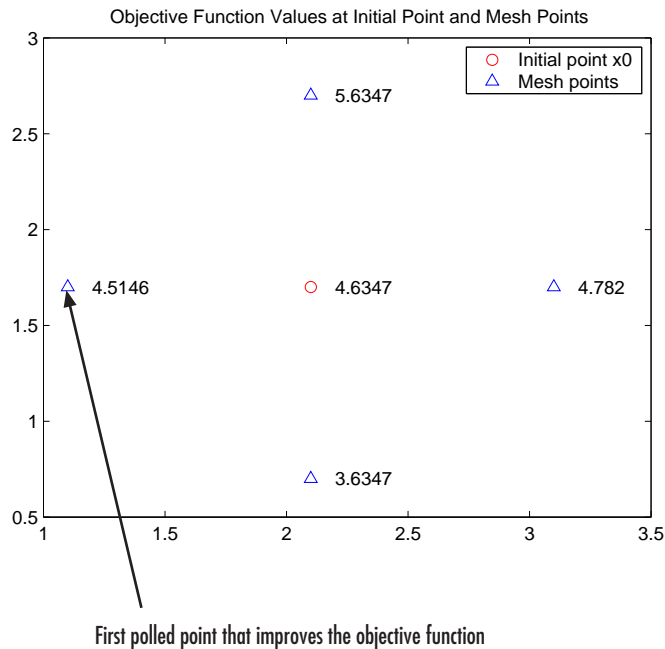
The algorithm polls the mesh points by computing their objective function values until it finds one whose value is smaller than 4.6347, the value at `x0`. In this case, the first such point it finds is `[1.1 1.7]`, at which the value of the objective function is `4.5146`, so the poll at iteration 1 is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

Objective Function Values at x1 and Mesh Points
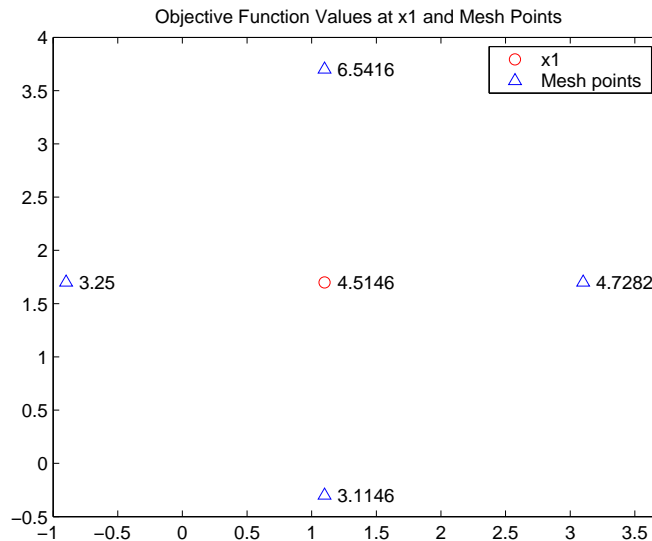
The algorithm polls the mesh points until it finds one whose value is smaller than 4.5146, the value at x1. The first such point it finds is [-0.9 1.7], at which the value of the objective function is 3.25, so the poll at iteration 2 is again successful. The algorithm sets the second point in the sequence 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
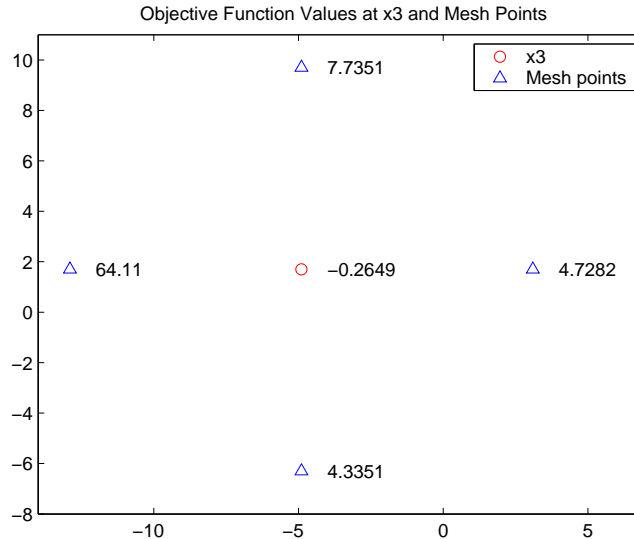
At this iteration, none of the mesh points has a smaller objective function value than the value at x3, so the poll is *unsuccessful*. In this case, the algorithm does not change the current point at the next iteration. That is,

```
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.

⊟ Display to command window

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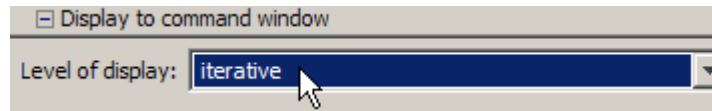
With this setting, the pattern search displays information about each iteration
at the command line. The first four iterations are

```
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
   4         14       -0.264905           4       Refine Mesh
```

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 4, the entry Refine Mesh 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 60 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
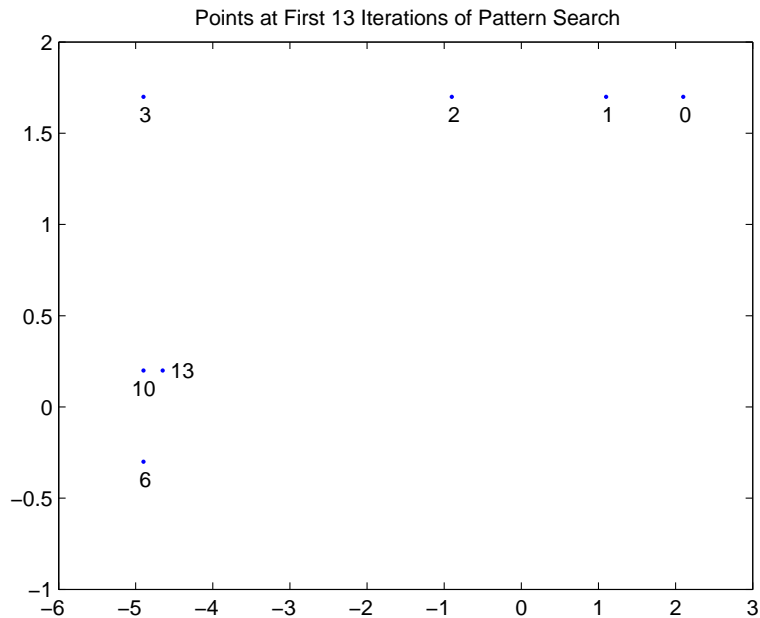
The numbers below the points indicate the first iteration at which the algorithm finds the point. The plot only shows iteration numbers corresponding to successful polls, because the best point doesn't change after an unsuccessful poll. For example, the best point at iterations 4 and 5 is the same as 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 an additional parameter called the poll parameter, Δ_p, in the mesh size stopping criterion:

$$\Delta_p = \begin{cases} N\sqrt{\Delta_m} & \text{for positive basis } N+1 \text{ poll} \\ \sqrt{\Delta_m} & \text{for positive basis } 2N \text{ poll,} \end{cases}$$

where Δ_m is the mesh size. The MADS stopping criterion is:

$\Delta_p \leq$ **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] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

[2] 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.

[3] 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.

Performing a Pattern Search Using the Optimization Tool GUI

In this section...
"Example: A Linearly Constrained Problem" on page 4-26
"Displaying Plots" on page 4-29
"Example: Working with a Custom Plot Function" on page 4-30

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

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

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
```

lincontest7 is a file included in Global Optimization 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 9; 1 -1 2 -2 3 -3];
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.
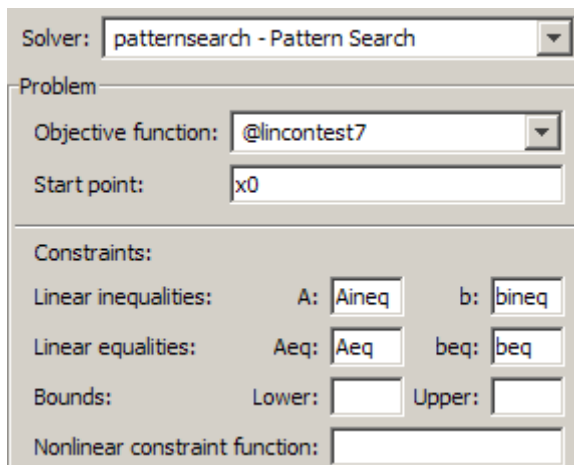
Since this is a linearly constrained problem, set the **Poll method** to GSS Positive basis 2N. For more information about the efficiency of the GSS search methods for linearly constrained problems, see "Example: Comparing the Efficiency of Poll Options" on page 4-48.
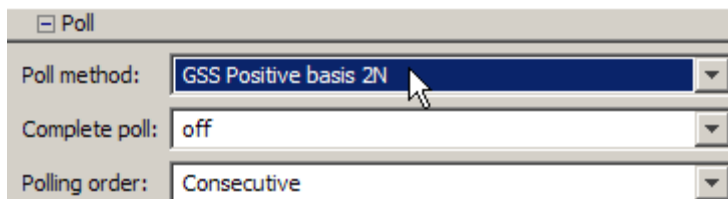
Then click **Start** to run the pattern search. When the search is finished, the results are displayed in **Run solver and view results** pane, as shown in the following figure.
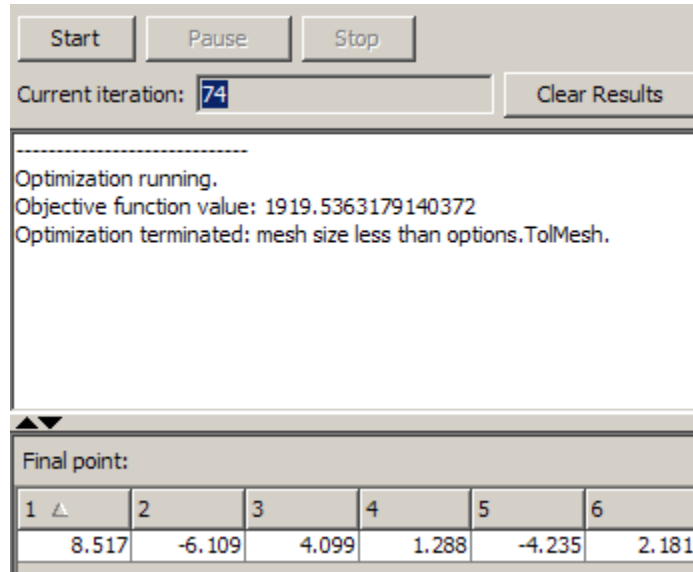
Displaying Plots

The **Plot functions** pane, shown in the following figure, enables you to display various plots of the results of a pattern search.
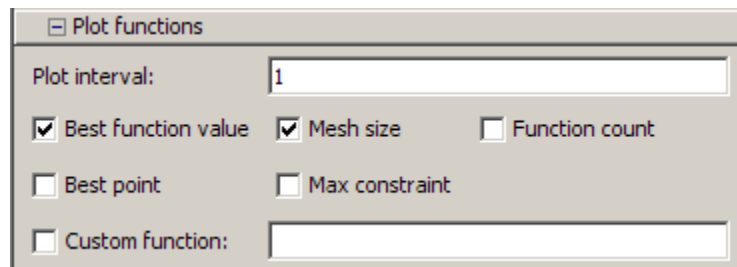
Select the check boxes next to the plots you want to display. For example, if you select **Best function value** and **Mesh size**, and run the example described in "Example: Finding the Minimum of a Function Using the GPS Algorithm" on page 4-7, the tool displays the plots shown in the following figure.
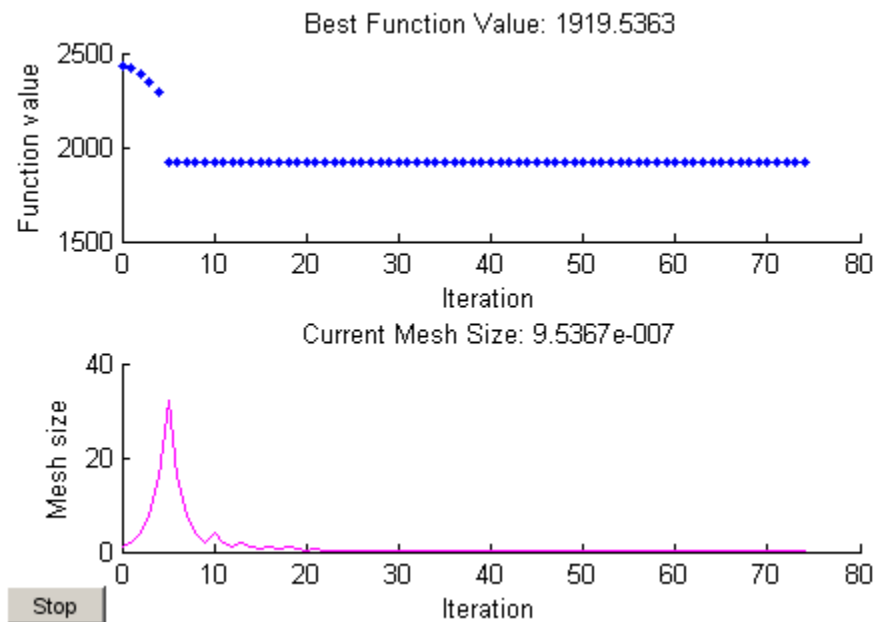
The upper plot displays the objective function value at each iteration. The lower plot displays the mesh size at each iteration.

Note When you display more than one plot, you can open a larger version of a plot in a separate window. Right-click (**Ctrl**-click for Mac) on a blank area in a plot while patternsearch is running, or after it has stopped, and choose the sole menu item.

"Plot Options" on page 9-30 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 4-31
- "Setting Up the Problem" on page 4-32
- "Using the Custom Plot Function" on page 4-33
- "How the Plot Function Works" on page 4-34

Creating the Custom Plot Function

To create the plot function for this example, copy and paste the following code into a new function 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 == 0
last_best = best;

else
```

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

Then save the file as psplotchange.m in a folder on the MATLAB path.

Setting Up the Problem

The problem is the same as "Example: A Linearly Constrained Problem" on page 4-26. To set up the problem:

1 Enter the following at the MATLAB command line:
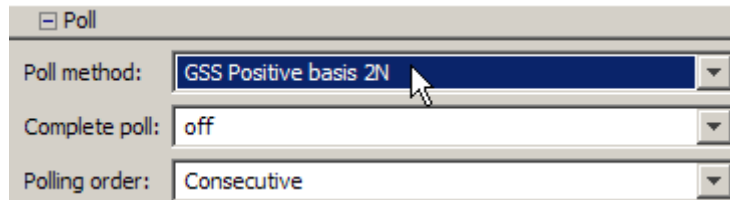
```
x0 = [2 1 0 9 1 0];
Aineq = [-8 7 3 -4 9 0];
bineq = 7;
Aeq = [7 1 8 3 3 3; 5 0 -5 1 -5 8; -2 -6 7 1 1 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

2 Enter optimtool to open the Optimization Tool.

3 Choose the patternsearch solver.

4 Set up the problem to match the following figure.
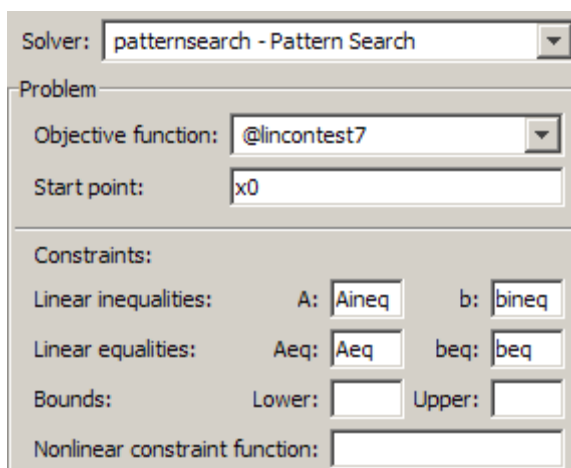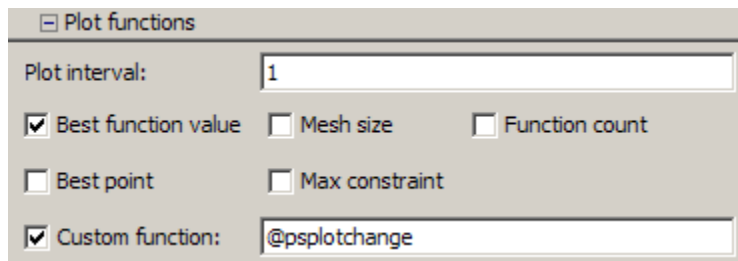
5 Since this is a linearly constrained problem, set the **Poll method** to GSS Positive basis 2N.

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, the pattern search tool displays the plots shown in the following figure.
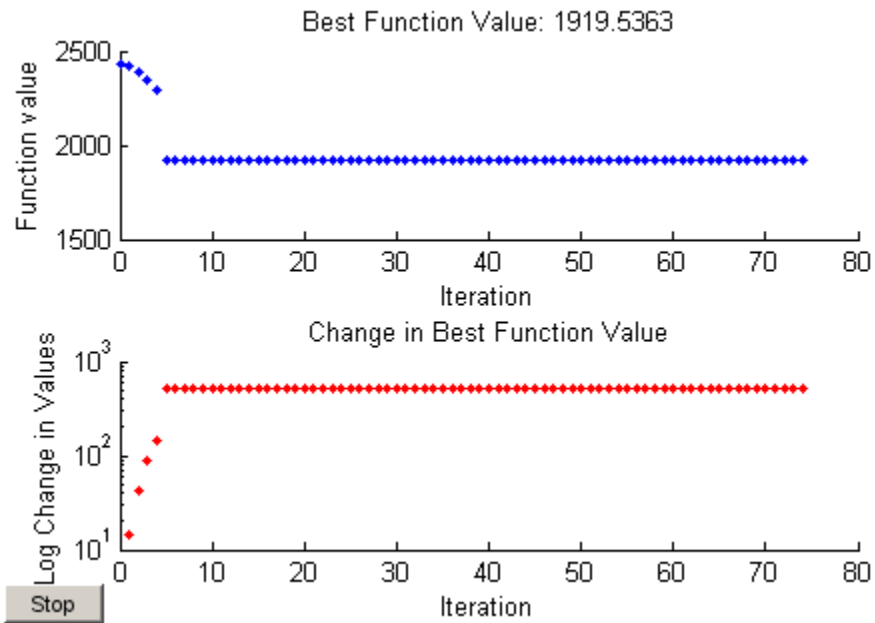
Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot will only show changes that are greater than 0.

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

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 9-9.
`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

In this section...

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 a function file. See "Computing Objective Functions" on page 2-2 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 4-7 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 $A\,x \leq b$.
- Aeq is a matrix and beq is a vector that represent equality constraints of the form $Aeq\,x = beq$.
- lb and ub are vectors representing bound constraints of the form $lb \leq x$ 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 patternsearch function 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 `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 empty except for `MeshExpansion`, which is set to 3. (An empty field causes `patternsearch` to use the default value.)

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 documentation. 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

Note All examples use the generalized pattern search (GPS) algorithm, but can be applied to the other algorithms 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 4-14 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, and similarly for GSS. For example, if you run a pattern search on the example described in "Example: A Linearly Constrained Problem" on page 4-26, the algorithm performs 1588 function evaluations with GPS Positive basis 2N, the default **Poll method**, but only 877 function evaluations using GPS Positive basis Np1. For more detail, see "Example: Comparing the Efficiency of Poll Options" on page 4-48.

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. In the Optimization Tool you can make the pattern search poll the entire mesh setting **Complete poll** to On in **Poll** options. At the command line, use psoptimset to set the CompletePoll option to 'on'.

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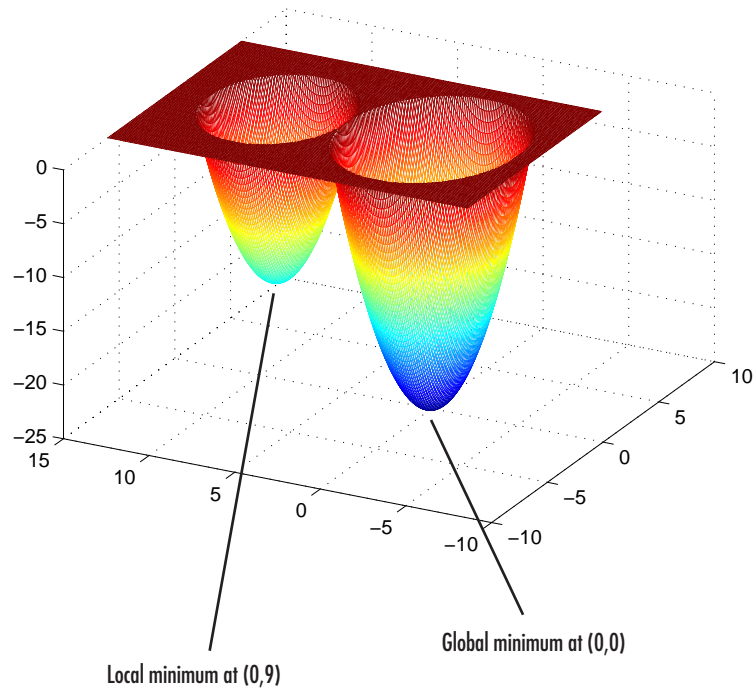
The global minimum of the function occurs at (0, 0), where its value is -25. However, the function also has a local minimum at (0, 9), where its value is -16.

To create a file that computes the function, copy and paste the following code into a new 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 folder 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.
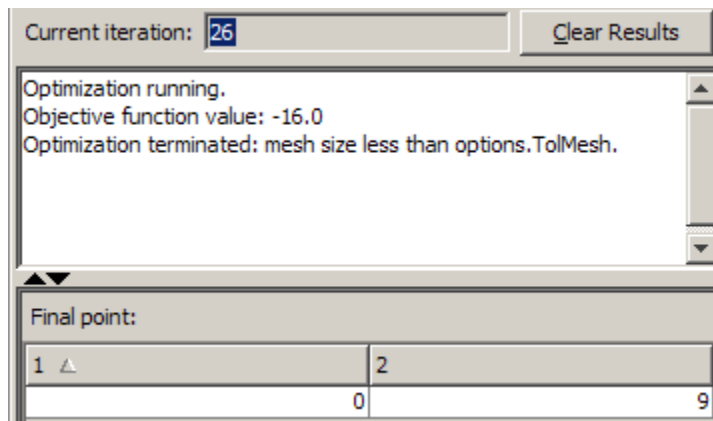
The pattern search returns the local minimum at (0, 9). At the initial point, (0, 5), the objective function value is 0. At the first iteration, the search polls the following mesh points.

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

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

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       f(x)       MeshSize       Method
   0          1           0            1
   1          3          -7            2       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.
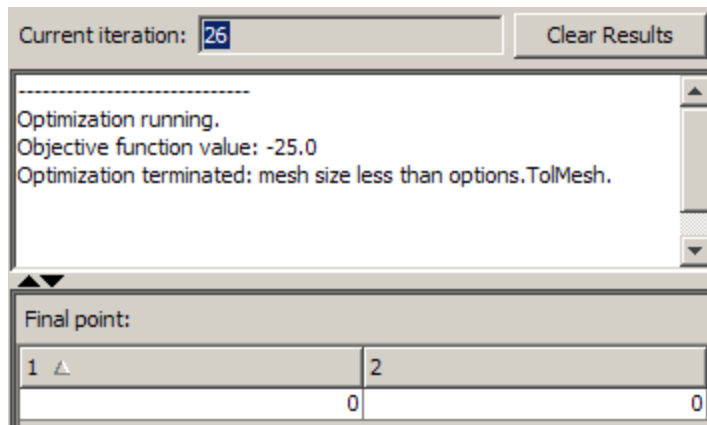
This time, the pattern search finds the global minimum at (0, 0). The difference between this run and the previous one is that with **Complete poll** set to On, at the first iteration the pattern search polls all four mesh points.

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

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

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

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

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      f(x)      MeshSize    Method
   0        1          0           1
   1        5         -9           2       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.
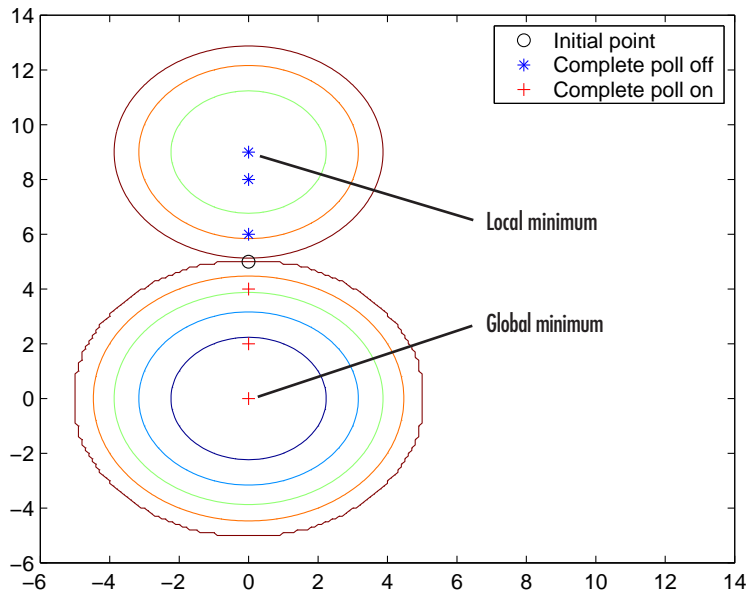
Example: Comparing the Efficiency of Poll Options

This example shows how several poll options interact in terms of iterations and total function evaluations. The main results are:

- GSS is more efficient than GPS or MADS for linearly constrained problems.

- Whether setting CompletePoll to 'on' increases efficiency or decreases efficiency is unclear, although it affects the number of iterations.

- Similarly, whether having a 2N poll is more or less efficient than having an Np1 poll is also unclear. The most efficient poll is GSS Positive Basis Np1 with **Complete poll** set to on. The least efficient is MADS Positive Basis Np1 with **Complete poll** set to on.

Note The efficiency of an algorithm depends on the problem. GSS is efficient for linearly constrained problems. However, predicting the efficiency implications of the other poll options is difficult, as is knowing which poll type works best with other constraints.
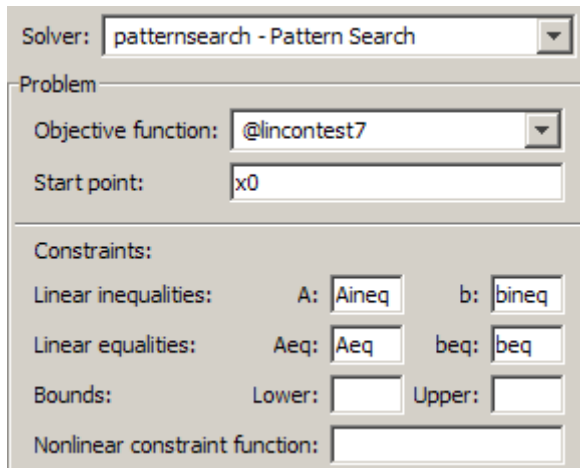
Problem setup

The problem is the same as in "Performing a Pattern Search on the Example" on page 4-27. This linearly constrained problem uses the lincontest7 file that comes with the toolbox:

1 Enter the following into your MATLAB workspace:

```
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 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

2 Open the Optimization Tool by entering optimtool at the command line.

3 Choose the patternsearch solver.

4 Enter the problem and constraints as pictured.
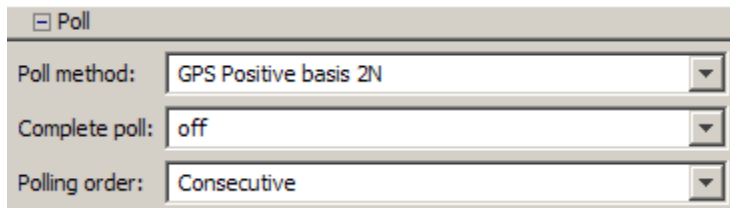
5 Ensure that the **Poll method** is GPS Positive basis 2N.
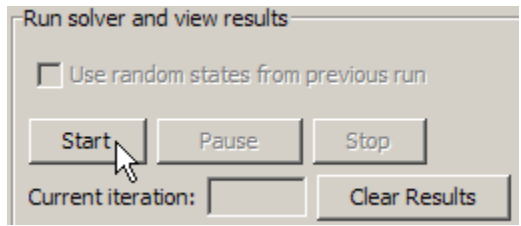
Generate the Results

1 Run the optimization.

2 Choose **File > Export to Workspace**.

3 Export the results to a structure named gps2noff.

4 Set **Options > Poll > Complete poll** to on.

5 Run the optimization.

6 Export the result to a structure named `gps2non`.

7 Set **Options > Poll > Poll method** to `GPS Positive basis Np1` and set **Complete poll** to `off`.

8 Run the optimization.

9 Export the result to a structure named `gpsnp1off`.

10 Set **Complete poll** to `on`.

11 Run the optimization.

12 Export the result to a structure named `gpsnp1on`.

13 Continue in a like manner to create solution structures for the other poll methods with **Complete poll** set on and off: `gss2noff`, `gss2non`, `gssnp1off`, `gssnp1on`, `mads2noff`, `mads2non`, `madsnp1off`, and `madsnp1on`.
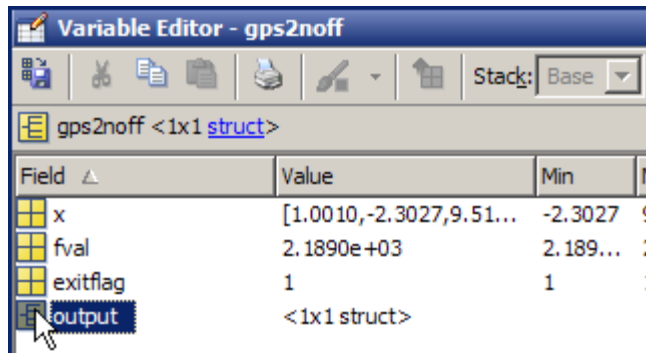
Examine the Results

You have the results of 12 optimization runs. The following table shows the efficiency of the runs, measured in total function counts and in iterations. Your MADS results could differ, since MADS is a stochastic algorithm.

Algorithm	Function Count	Iterations
GPS2N, complete poll off	1206	106
GPS2N, complete poll on	1362	94
GPSNp1, complete poll off	914	122
GPSNp1, complete poll on	991	106
GSS2N, complete poll off	683	74
GSS2N, complete poll on	889	74
GSSNp1, complete poll off	529	94
GSSNp1, complete poll on	519	74
MADS2N, complete poll off	907	159
MADS2N, complete poll on	1292	160
MADSNp1, complete poll off	1215	208
MADSNp1, complete poll on	1714	207

To obtain, say, the first row in the table, enter `gps2noff.output.funccount` and `gps2noff.output.iterations`. You can also examine a structure in the Variable Editor by double-clicking the structure in the Workspace Browser, and then double-clicking the `output` structure.
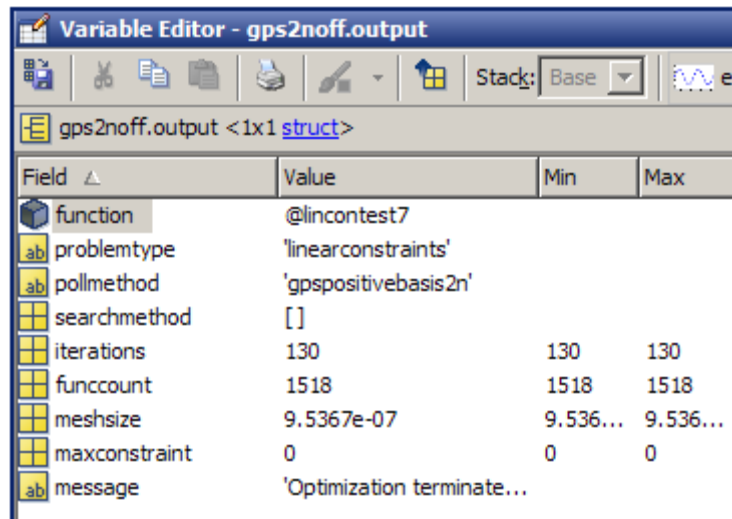
The main results gleaned from the table are:

- Setting **Complete poll** to on generally lowers the number of iterations for GPS and GSS, but the change in number of function evaluations is unpredictable.

- Setting **Complete poll** to on does not change the number of iterations for MADS, but substantially increases the number of function evaluations.

- The most efficient algorithm/options settings, with efficiency meaning lowest function count:

1 `GSS Positive basis Np1` with **Complete poll** set to `on` (function count 519)

2 `GSS Positive basis Np1` with **Complete poll** set to `off` (function count 529)

3 `GSS Positive basis 2N` with **Complete poll** set to `off` (function count 683)

4 `GSS Positive basis 2N` with **Complete poll** set to `on` (function count 889)

The other poll methods had function counts exceeding 900.

- For this problem, the most efficient poll is `GSS Positive Basis Np1` with **Complete poll** set to `on`, although the **Complete poll** setting makes only a small difference. The least efficient poll is `MADS Positive Basis Np1` with **Complete poll** set to `on`. In this case, the **Complete poll** setting makes a substantial difference.

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 4-26. 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 9; 1 -1 2 -2 3 -3];
beq = [84 62 65 1];
```

Then enter the settings shown in the following figures in the Optimization Tool.

Solver: | patternsearch - Pattern Search ▼

Problem

Objective function: | @lincontest7 ▼

Start point: | x0

Constraints:

Linear inequalities: A: | Aineq | b: | bineq

Linear equalities: Aeq: | Aeq | beq: | beq

Bounds: Lower: | | Upper: | |

Nonlinear constraint function: | |

☐ Poll

Poll method: | GPS Positive basis 2N ▼

Complete poll: | off ▼

Polling order: | Consecutive ▼

☐ Plot functions

Plot interval: | 1

☑ Best function value ☐ Mesh size ☑ Function count

☐ Best point ☐ Max constraint

☐ Custom function: | |

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

Best Function Value: 1919.4867

Total Function Evaluations: 1206
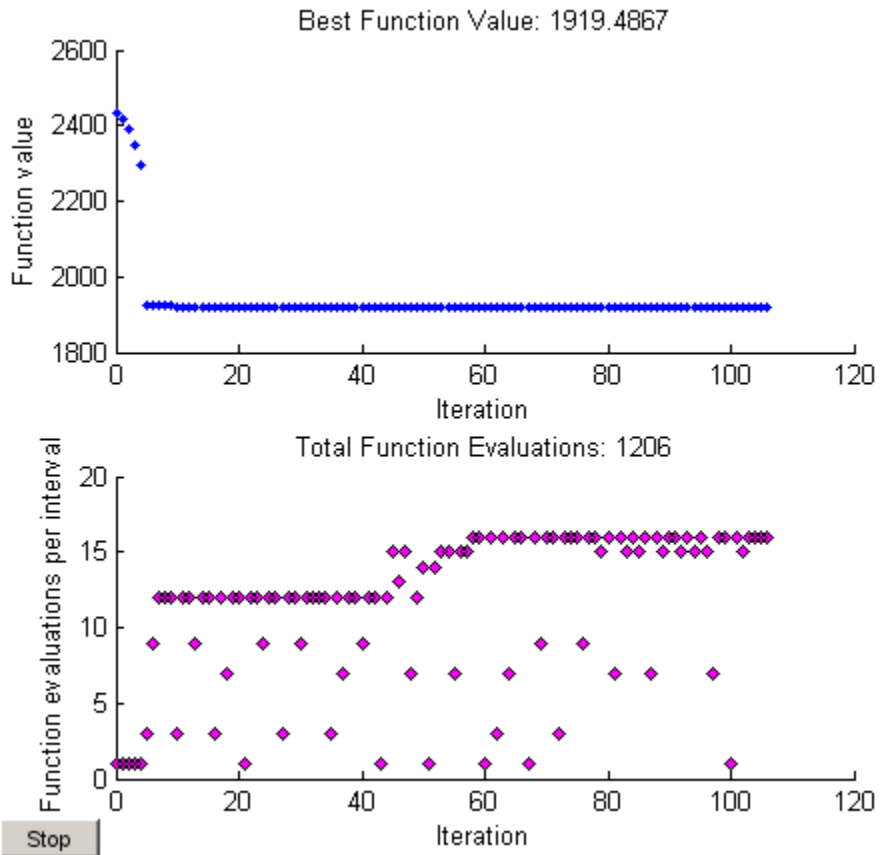
To see the effect of using a search method, select MADS Positive Basis 2N in the **Search method** field in **Search** options.
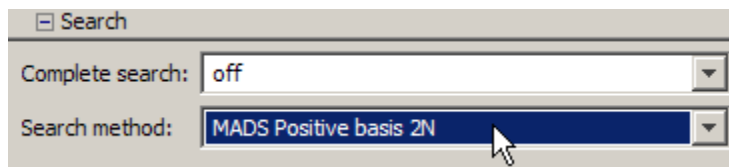
This sets the search method to be a pattern search using the pattern for MADS Positive Basis 2N. Then click **Start** to run the pattern search. This displays the following plots.
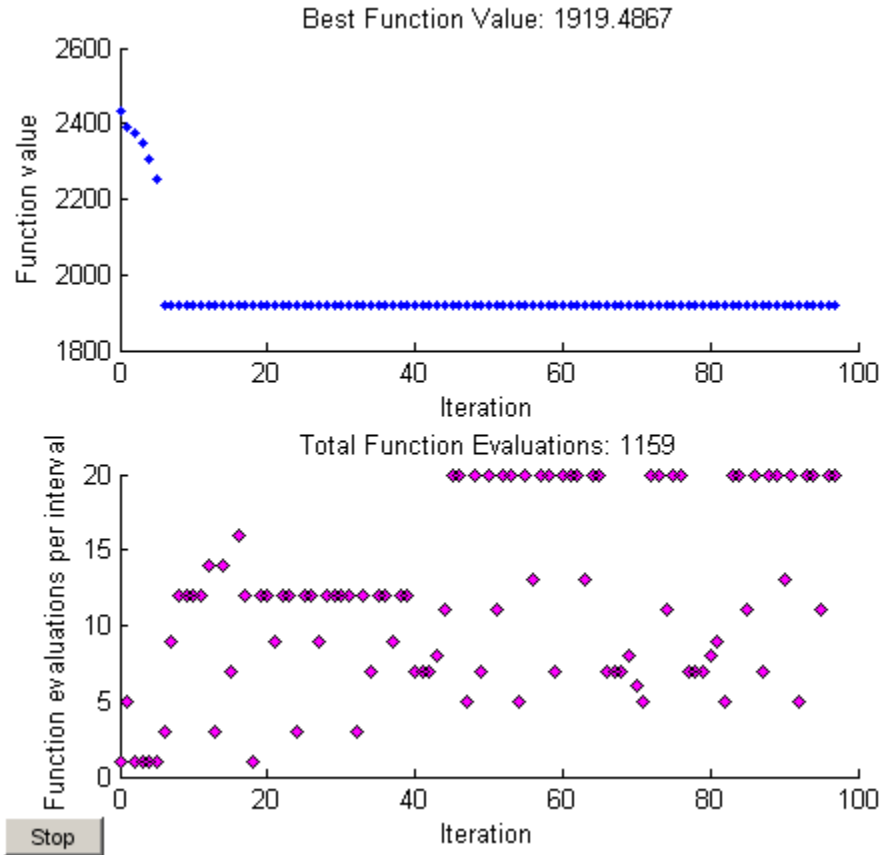
Note that using the search method reduces the total function evaluations—from 1206 to 1159—and reduces the number of iterations from 106 to 97.

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.
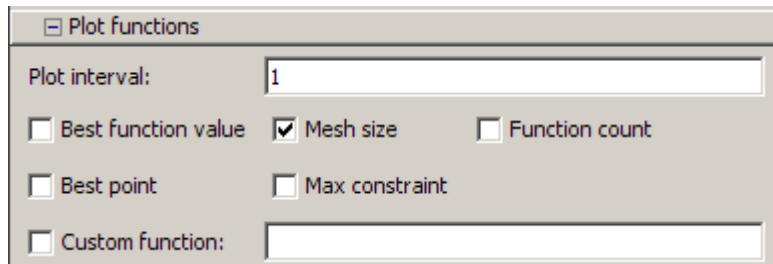
To also display the values of the mesh size and objective function at the command line, set **Level of display** to Iterative in the **Display to command window** options.
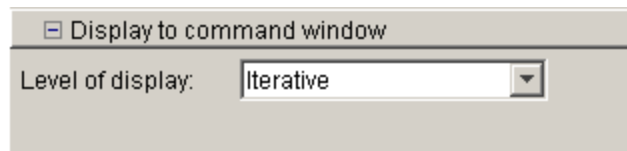
When you run the example described in "Example: A Linearly Constrained Problem" on page 4-26, the Optimization Tool displays the following plot.
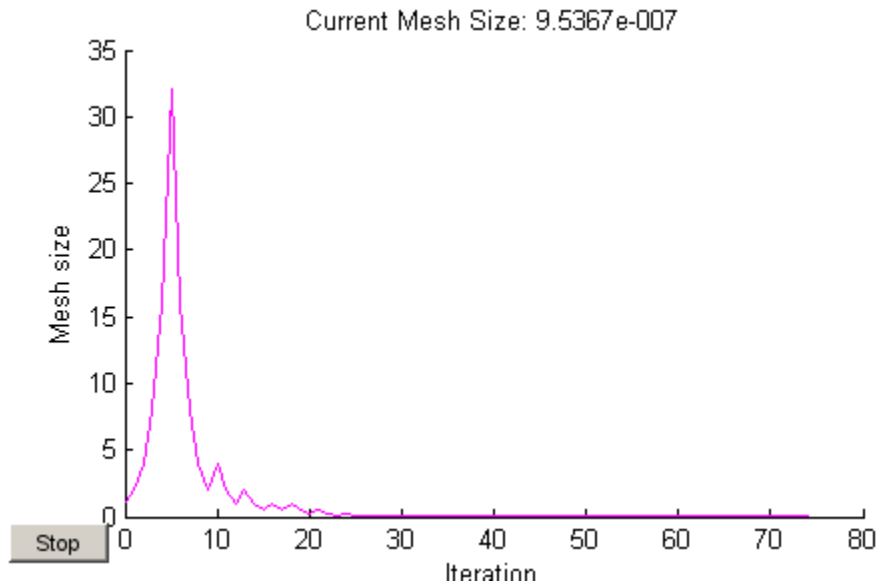
To see the changes in mesh size more clearly, change the *y*-axis to logarithmic scaling as follows:

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.
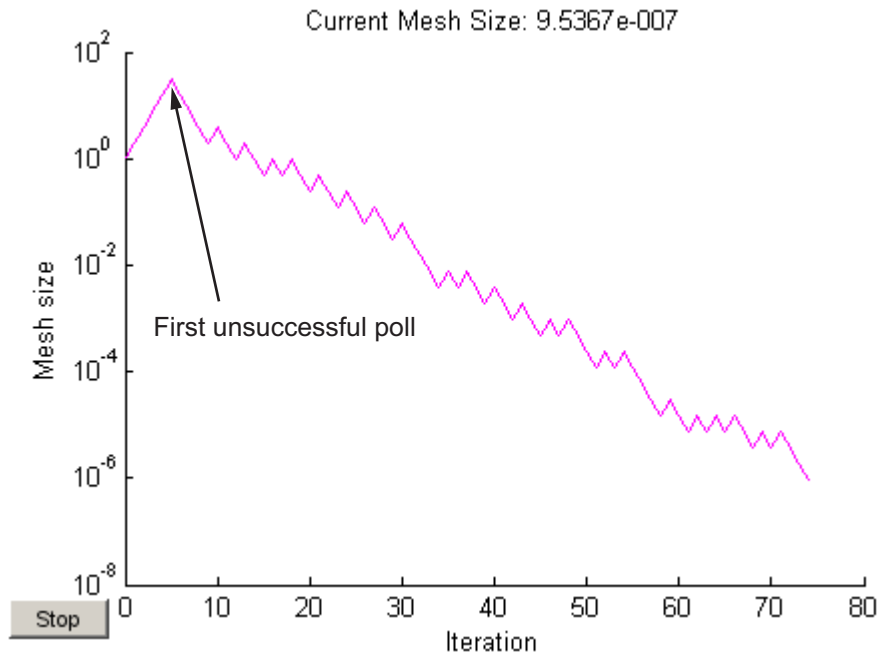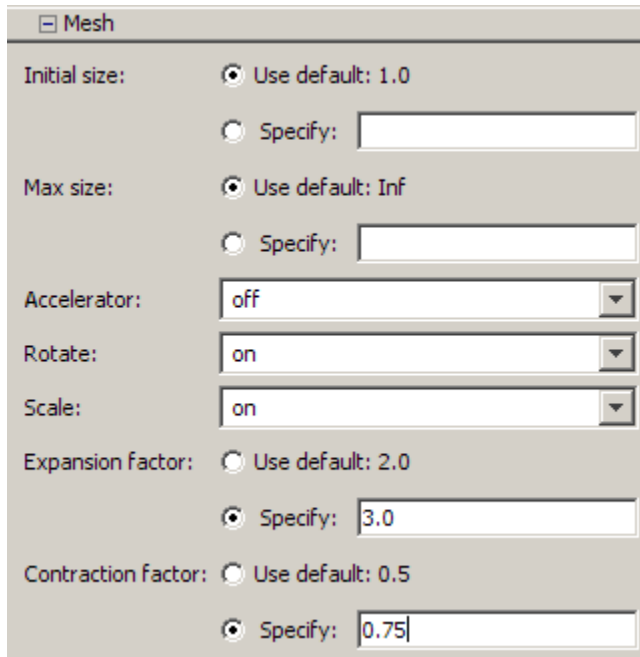
The first 5 iterations result in successful polls, so the mesh sizes increase steadily during this time. You can see that the first unsuccessful poll occurs at iteration 6 by looking at the command-line display:

```
Iter     f-count            f(x)       MeshSize        Method
   0         1          2434.15              1
   1         2          2419.37              2       Successful Poll
   2         3          2392.12              4       Successful Poll
   3         4          2346.85              8       Successful Poll
   4         5          2293.23             16       Successful Poll
   5         8          1921.94             32       Successful Poll
   6        17          1921.94             16       Refine Mesh
```

Note that at iteration 5, which is successful, the mesh size doubles for the next iteration. But at iteration 6, 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.
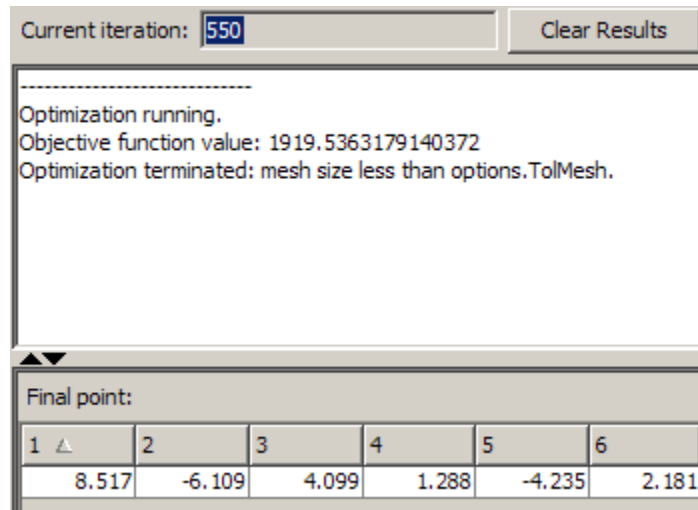
When you change the scaling of the *y*-axis to logarithmic, the mesh size plot appears as shown in the following figure.
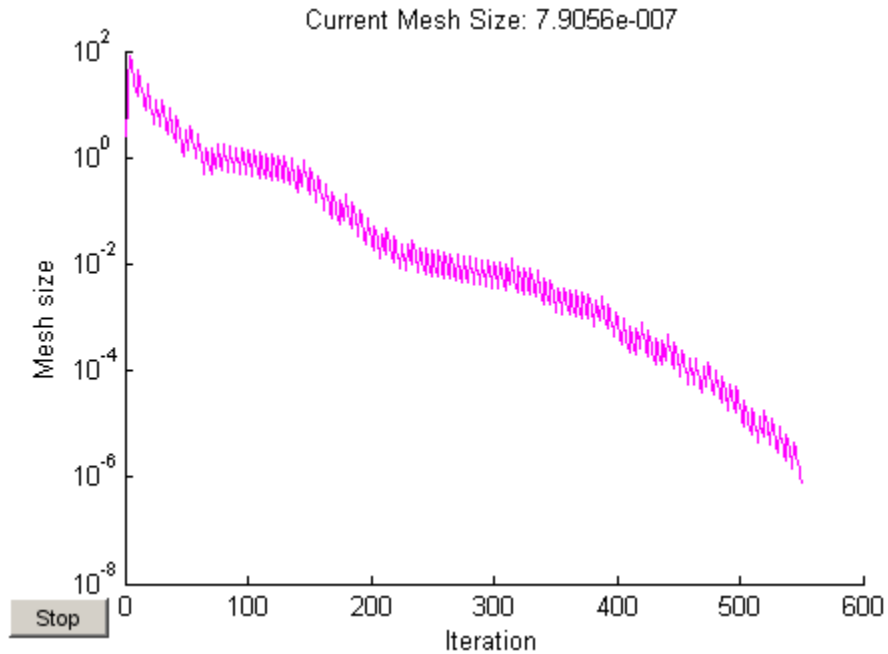
Note that the mesh size increases faster with **Expansion factor** set to 3.0, as compared with the default value of 2.0, and decreases more slowly with **Contraction factor** set to 0.75, as compared with the default value of 0.5.

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 4-26, the number of iterations required to reach the mesh tolerance is 61, as compared with 74 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 56, but differ at iteration 57. The MATLAB Command Window displays the following lines for iterations 56 and 57 with **Accelerator** set to Off.

```
Iter      f-count        f(x)       MeshSize       Method
  56        509        1919.54     6.104e-005    Refine Mesh
  57        521        1919.54     3.052e-005    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        f(x)       MeshSize       Method
  56        509        1919.54     6.104e-005    Refine Mesh
  57        521        1919.54     1.526e-005    Refine Mesh
```

In this case the mesh size is multiplied by 0.25.

Using Cache

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

You can eliminate these redundant computations by using a cache, that is, by storing a history of the points that the pattern search has already visited. To do so, set **Cache** to On in **Cache** options. At each poll, the pattern search checks to see whether the current mesh point is within a specified tolerance,

Tolerance, of a point in the cache. If so, the search does not compute the objective function for that point, but uses the cached function value and moves on to the next point.

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

To illustrate this, select **Best function value** and **Function count** in the **Plot functions** pane and run the example described in "Example: A Linearly Constrained Problem" on page 4-26 with **Cache** set to Off. After the pattern search finishes, the plots appear as shown in the following figure.
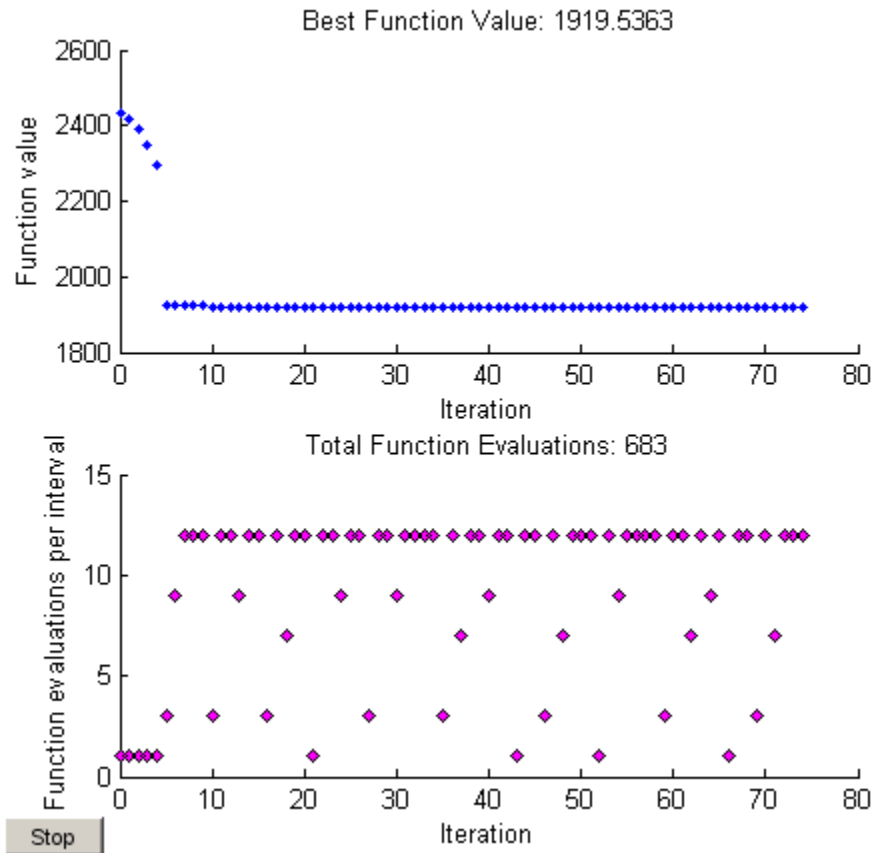
Note that the total function count is 683.

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

Best Function Value: 1919.5363

Total Function Evaluations: 664
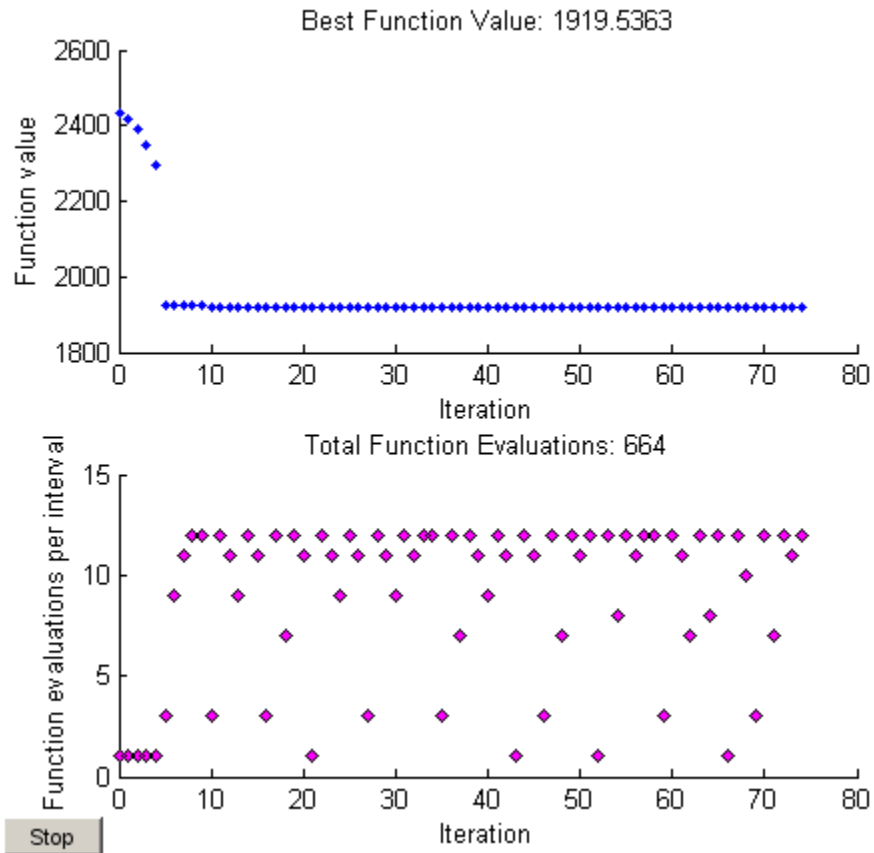
This time, the total function count is reduced to 664.

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 linearly constrained problems. It 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**.

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

$$
\begin{aligned}
&x_1 x_2 + x_1 - x_2 + 1.5 \le 0 \quad \text{(nonlinear constraint)} \\
&10 - x_1 x_2 \le 0 \qquad\qquad\;\; \text{(nonlinear constraint)} \\
&0 \le x_1 \le 1 \qquad\qquad\quad\; \text{(bound)} \\
&0 \le x_2 \le 13 \qquad\qquad\;\; \text{(bound)}
\end{aligned}
$$

Begin by creating the objective and constraint functions. First, create a 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 a 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
```
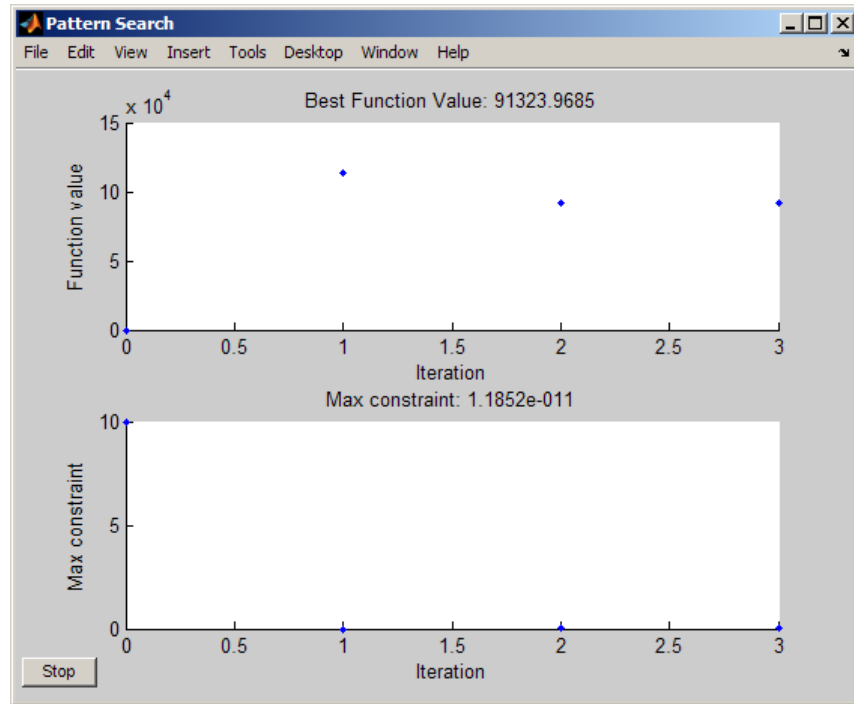
Best Objective Function Value and Maximum Constraint Violation at Each Iteration

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.
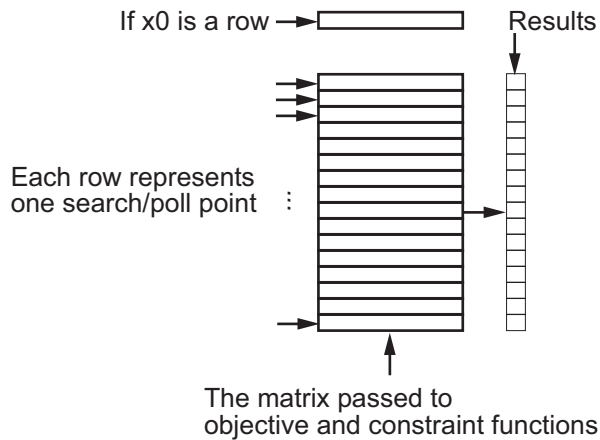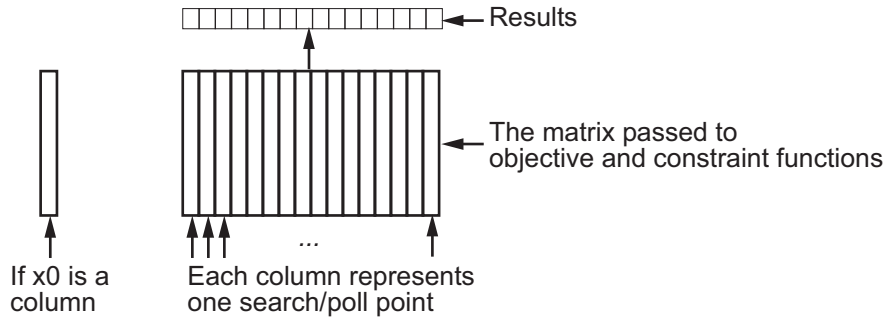
Structure of Vectorized Functions

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], a function 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], a function 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 4-74 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 4-74, "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 a function 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 4-73) 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
```

5

Using 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 a file

"Using the Genetic Algorithm from the Command Line" on page 5-39 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.
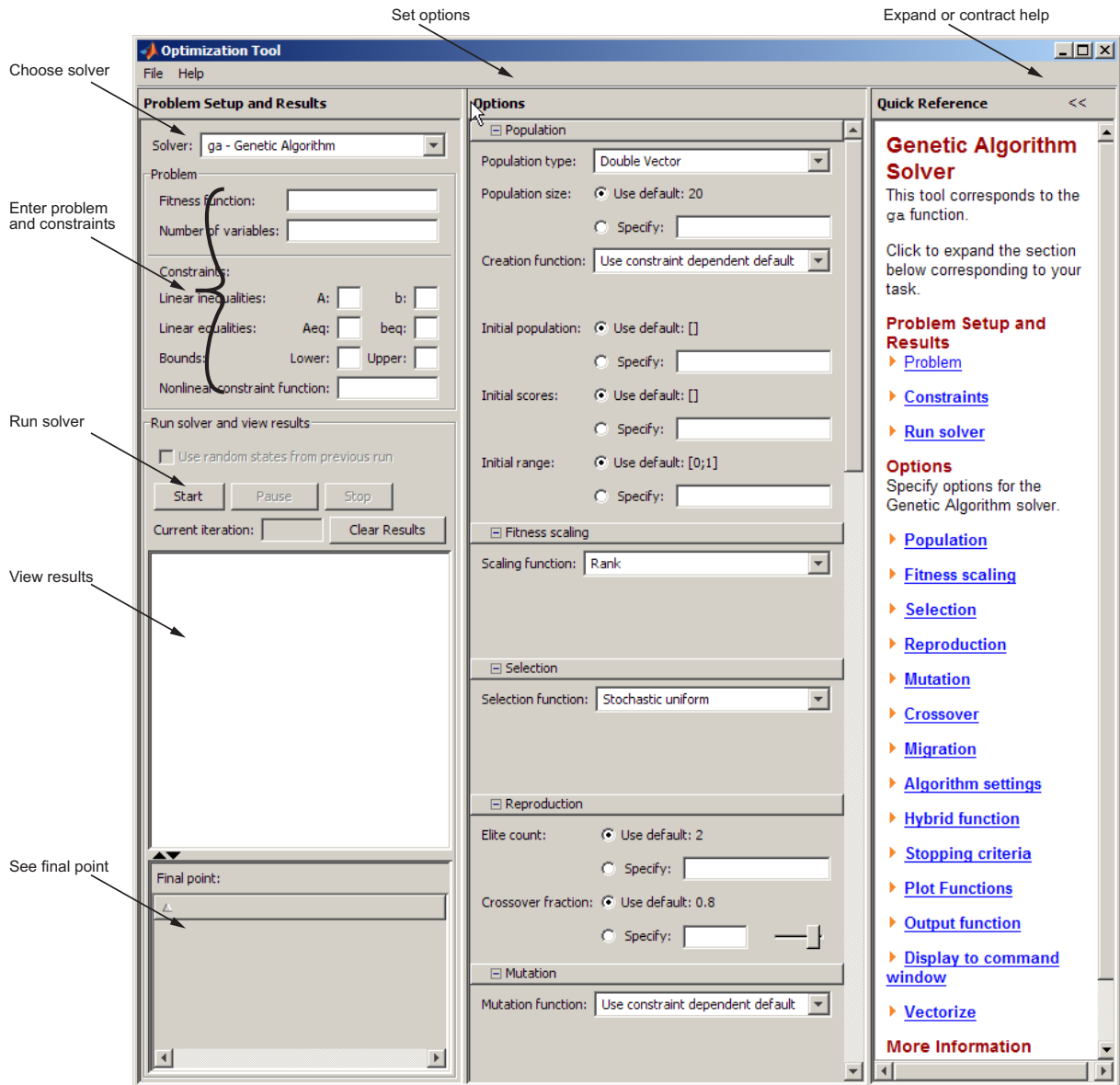
Set options

Expand or contract help

Choose solver

Enter problem and constraints

Run solver

View results

See final point

You can also start 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 a file that computes the fitness function. "Computing Objective Functions" on page 2-2 explains how write this 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 "Computing Objective Functions" on page 2-2, 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 documentation.
- See "Example: Rastrigin's Function" on page 5-8 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

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

Global Optimization Toolbox software contains the `rastriginsfcn.m` file, which computes the values of Rastrigin's function. The following figure shows a plot of Rastrigin's function.

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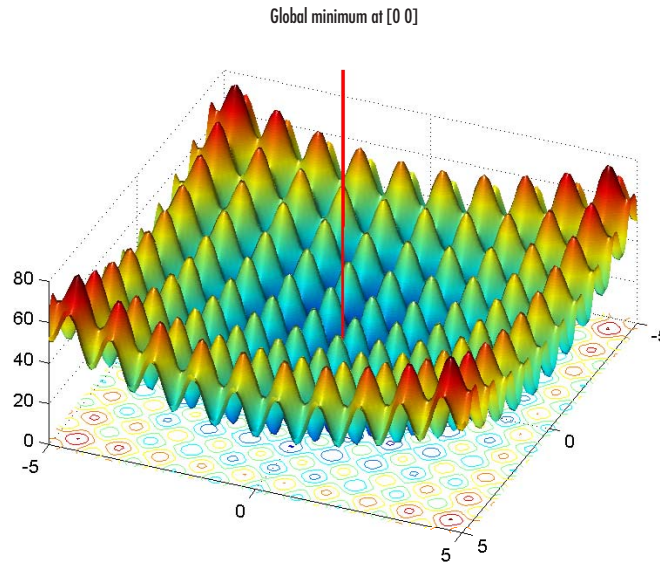
As the plot shows, Rastrigin's function has many local minima—the "valleys" in the plot. However, the function has just one global minimum, which occurs at the point [0 0] in the *x-y* plane, as indicated by the vertical line in the plot, where the value of the function is 0. At any local minimum other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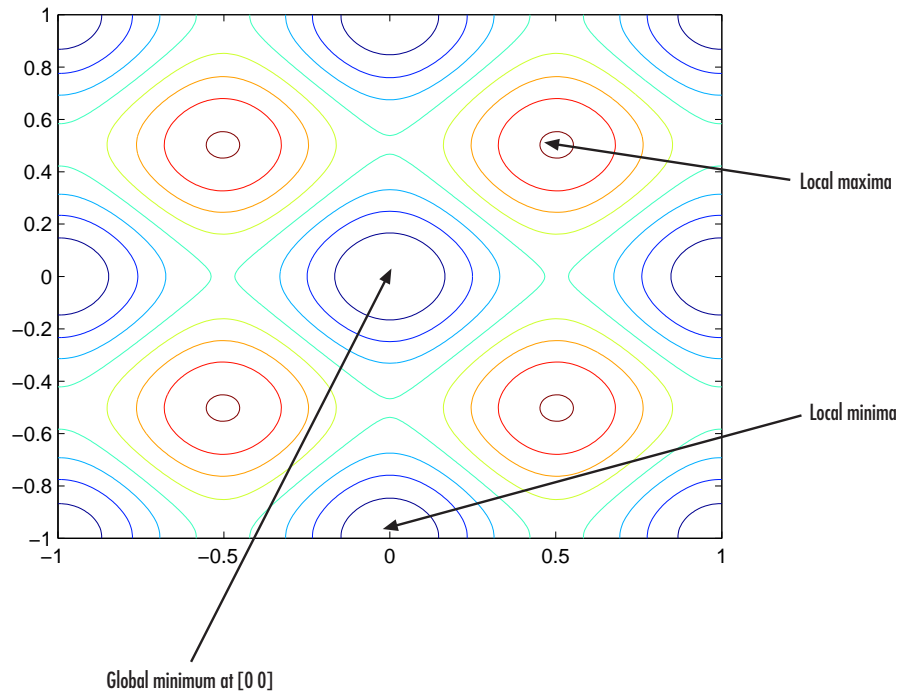
Finding the Minimum of Rastrigin's Function

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.
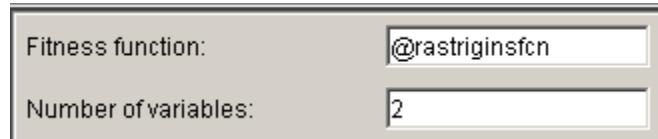
To find the minimum, do the following steps:

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

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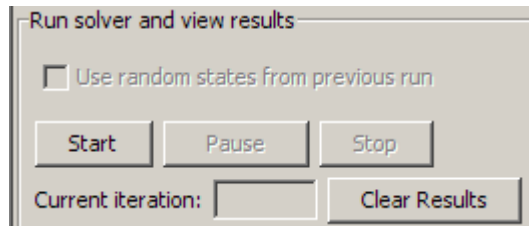
- In the **Fitness function** field, enter @rastriginsfcn.

- 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.

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:

Current iteration: 53 | Clear Results

```
--------------------------
Optimization running.
Objective function value: 0.05531602101322264
Optimization terminated: average change in the fitness value less
than options.TolFun.
```

Final point:

1 △	2
0.001	-0.017
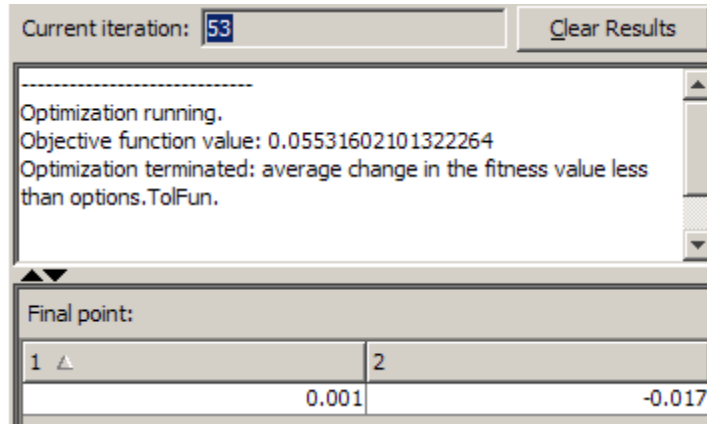
- The final value of the fitness function when the algorithm terminated:

  ```
  Objective function value: 0.05531602101322264
  ```

  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 5-49 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 [0.001 -0.017].

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.
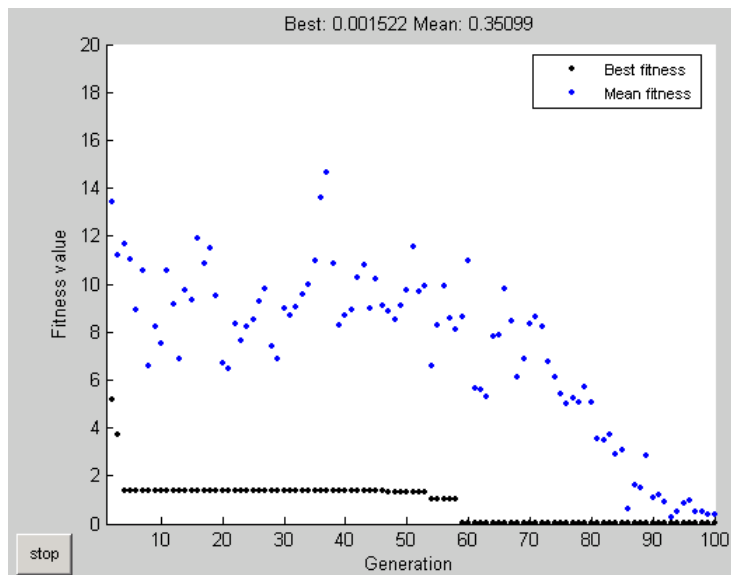
The points at the bottom of the plot denote the best fitness values, while the points above them denote the averages of the fitness values in each generation. The plot also displays the best and mean values in the current generation numerically at the top.

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.
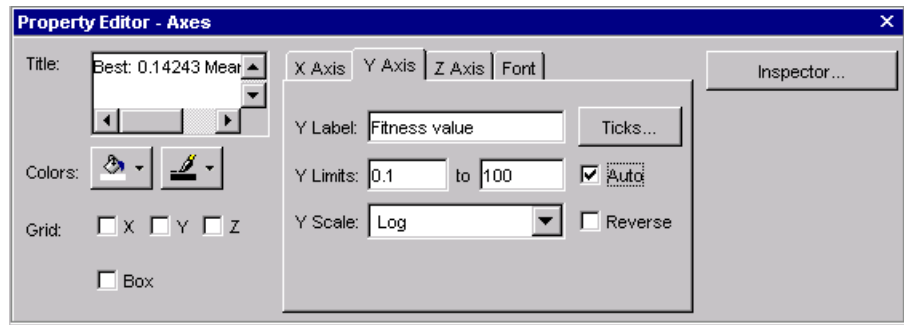
2 Click the **Y Axis** tab.

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

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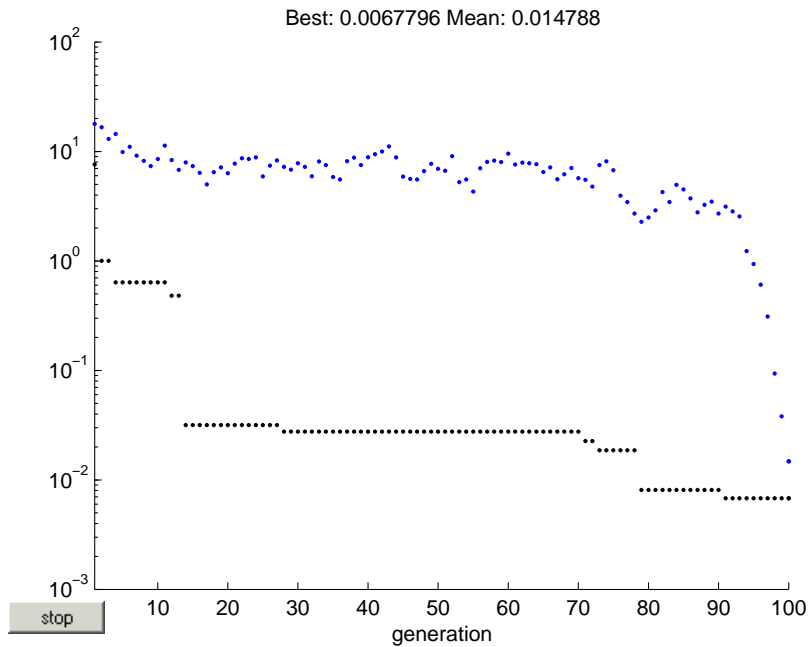
Typically, the best fitness value improves rapidly in the early generations, when the individuals are farther from the optimum. The best fitness value improves more slowly in later generations, whose populations are closer to the optimal point.

Some Genetic Algorithm Terminology

In this section...
"Fitness Functions" on page 5-17
"Individuals" on page 5-17
"Populations and Generations" on page 5-18
"Diversity" on page 5-18
"Fitness Values and Best Fitness Values" on page 5-19
"Parents and Children" on page 5-19

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.

Write the fitness function as a file or anonymous function, 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.
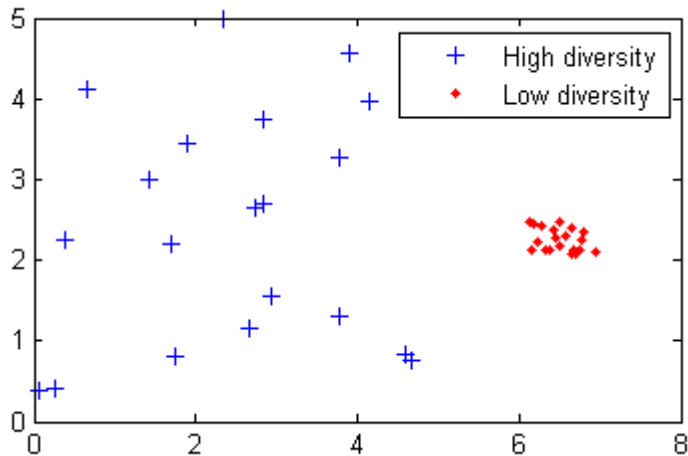
Diversity is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

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 5-20
"Initial Population" on page 5-21
"Creating the Next Generation" on page 5-22
"Plots of Later Generations" on page 5-24
"Stopping Conditions for the Algorithm" on page 5-24

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 5-24.

Initial Population

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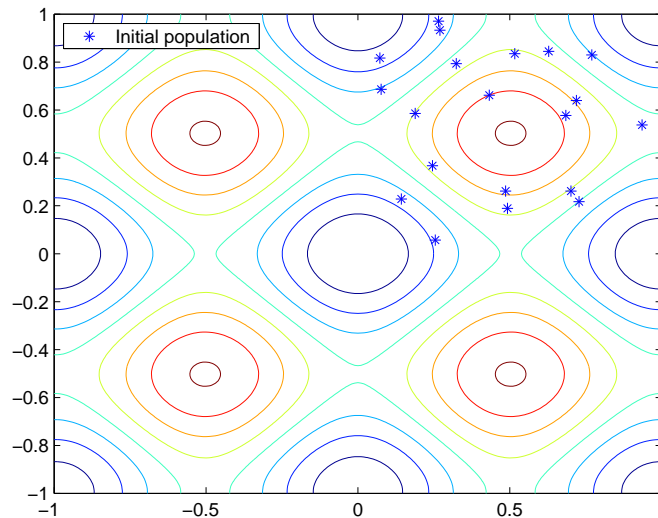
In this example, the initial population contains 20 individuals, which is the default value of **Population size** in the **Population** options. Note that all the individuals in the initial population lie in the upper-right quadrant of the picture, that is, their coordinates lie between 0 and 1, because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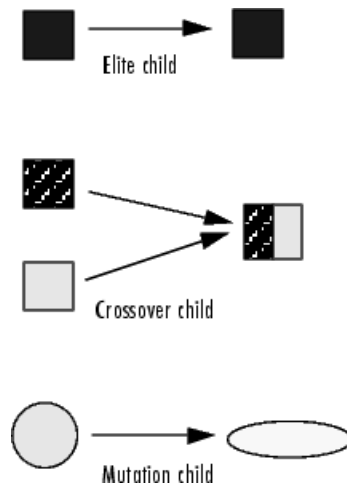
Mutation child

"Mutation and Crossover" on page 5-63 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.
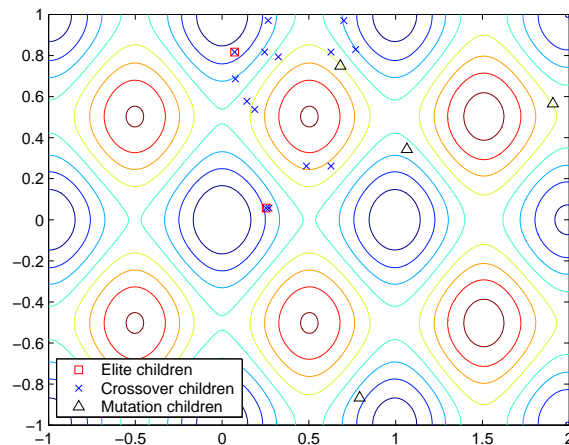
Plots of Later Generations

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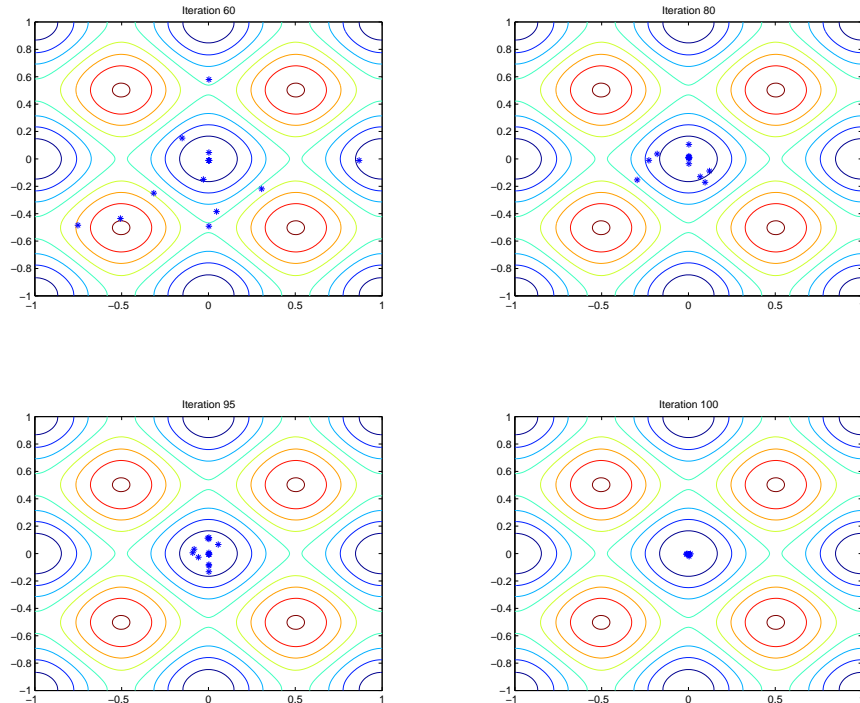
As the number of generations increases, the individuals in the population get closer together and approach the minimum point `[0 0]`.

Stopping Conditions for the Algorithm
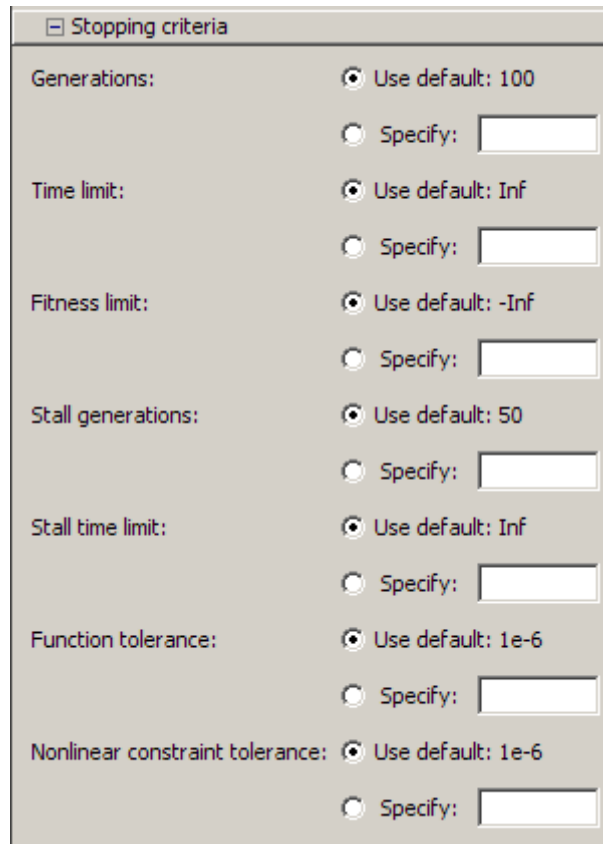
The genetic algorithm uses the following conditions to determine when to stop:

- **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:

[1] 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.

[2] 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.

Genetic Algorithm Optimizations Using the Optimization Tool GUI

Introduction

The Optimization Tool GUI is described in the chapter Optimization Tool in the Optimization Toolbox documentation. 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 5-8, the tool displays plots similar to those shown in the following figure.
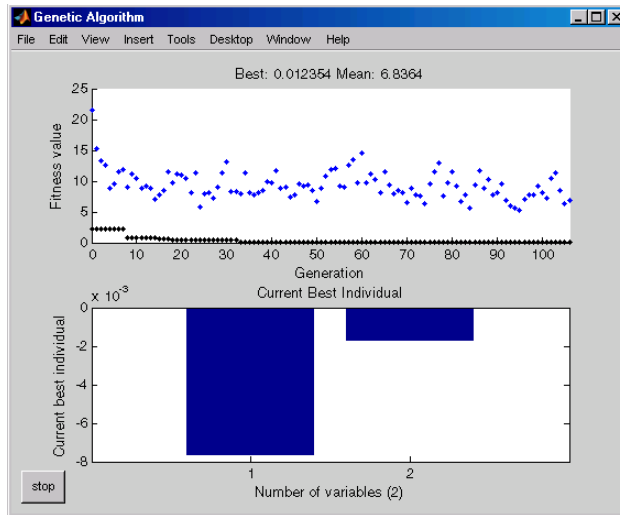
The upper plot displays the best and mean fitness values in each generation. The lower plot displays the coordinates of the point with the best fitness value in the current generation.

Note When you display more than one plot, you can open a larger version of a plot in a separate window. Right-click (**Ctrl**-click for Mac) on a blank area in a plot while ga is running, or after it has stopped, and choose the sole menu item.

"Plot Options" on page 9-30 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 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 file as gaplotchange.m in a folder 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 5-8, the tool displays plots similar to those shown in the following figure.
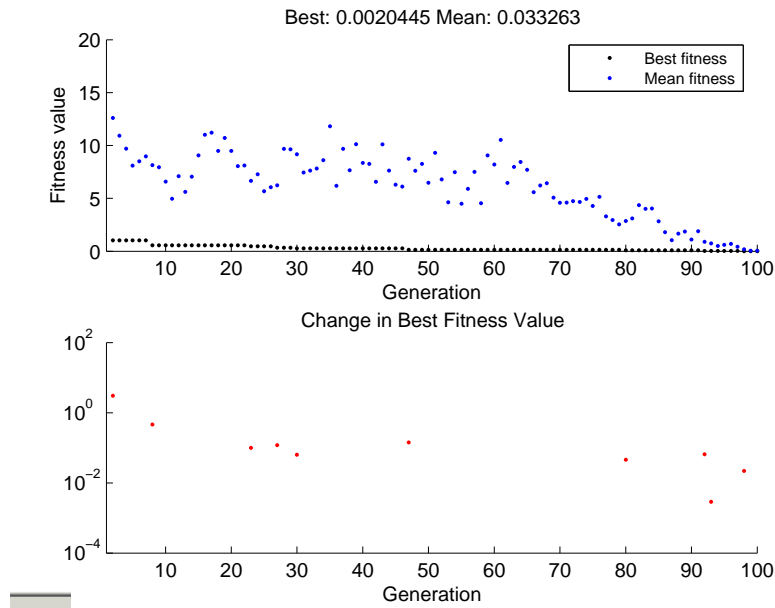
Note that because the scale of the *y*-axis in the lower custom plot is logarithmic, the plot only shows changes that are greater then 0. The logarithmic scale enables you to see small changes in the fitness function that the upper plot does not reveal.

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 9-32.

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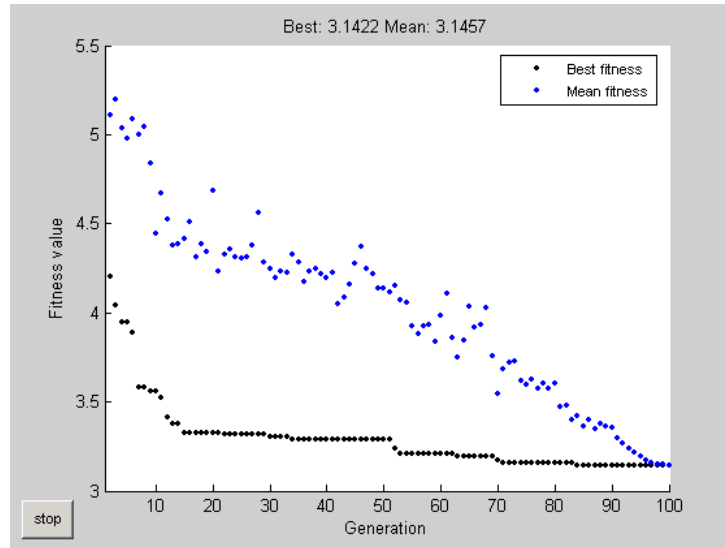
Suppose you want to experiment by running the genetic algorithm with other options settings, and then later restart this run from its final population with its current options settings. You can do this using the following steps:

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.
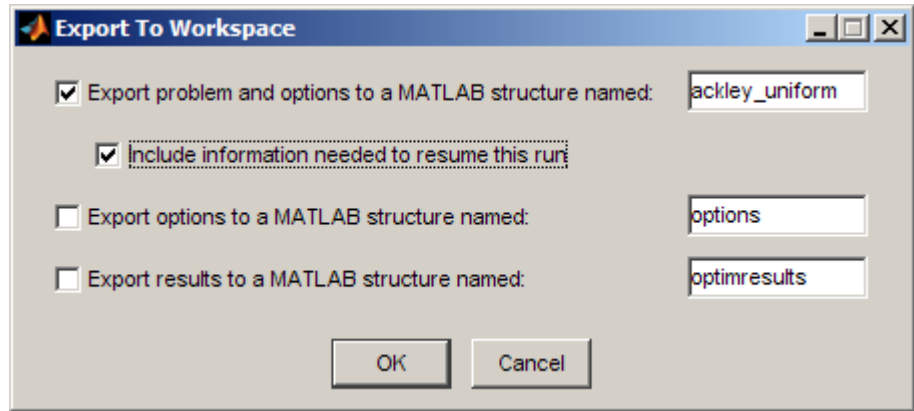
3 Click **OK**.

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: []
      rngstate: []
        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.
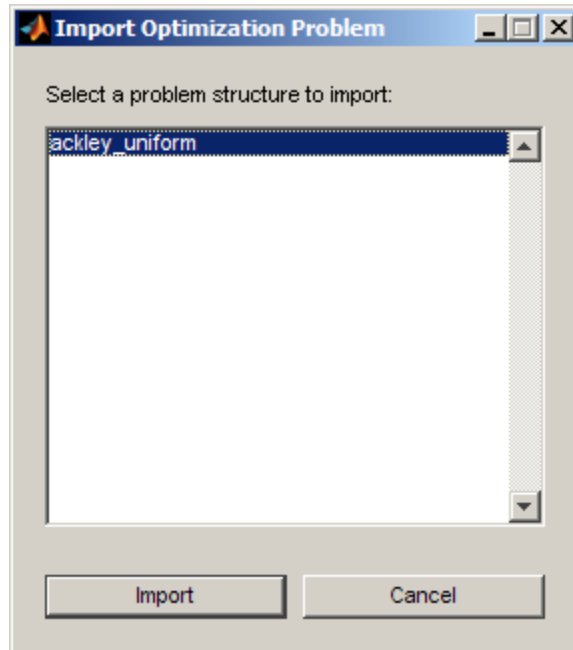
2 Select ackley_uniform.

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.
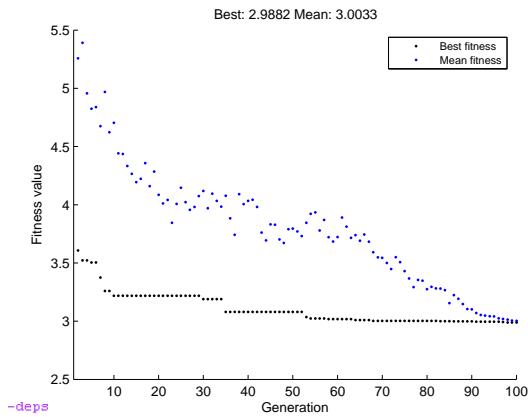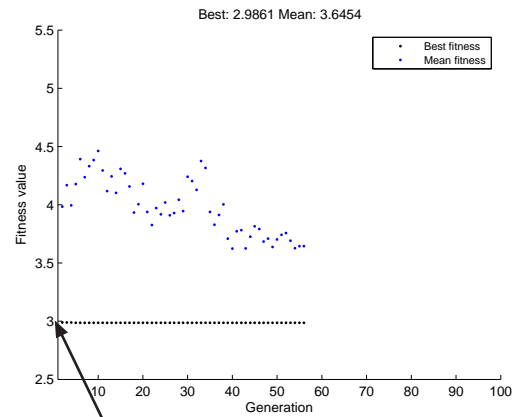
First run   Run resumes here

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 file that computes the fitness function. "Computing Objective Functions" on page 2-2 explains how to write this 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 5-8 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 default MATLAB pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time `ga` calls the stream, its state changes. So that the next time `ga` calls the stream, it returns a different random number. 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 state of the default stream, and then reset the state to this value 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 state of the stream is stored in `output.rngstate`:

```
output =
    problemtype: 'unconstrained'
       rngstate: [1x1 struct]
    generations: 68
      funccount: 1380
        message: 'Optimization terminated: average change in
                  the fitness value less than options.TolFun.'
```

To reset the state, enter

```
stream = RandStream.getDefaultStream;
stream.State = output.rngstate.state;
```

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.

```
┌Run solver and view results──────────────────┐
│                                             │
│   ☑ Use random states from previous run     │
│                                             │
└─────────────────────────────────────────────┘
```
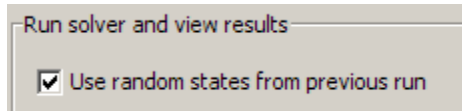
Note If you do not need to reproduce your results, it is better not to set the state of the stream, so that you get the benefit of the randomness in the genetic algorithm.

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.
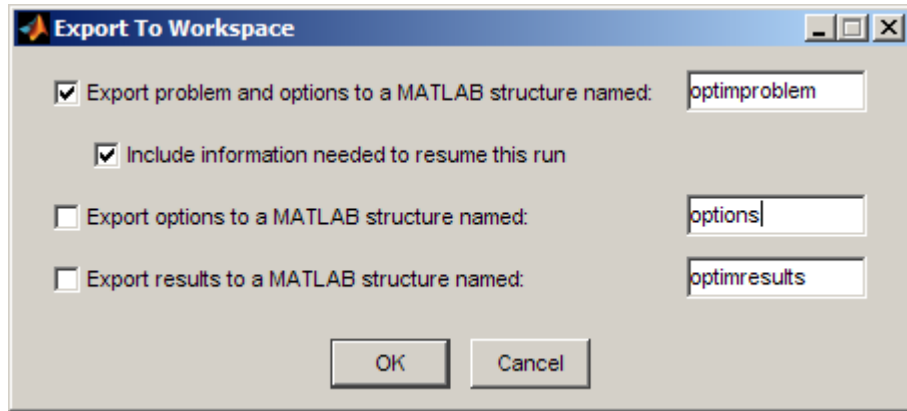
This saves the final population, which becomes the initial population when imported.

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 a File

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

```
options = gaoptimset('Generations',300);
strm = RandStream('mt19937ar','Seed',6525);
RandStream.setDefaultStream(strm);
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.
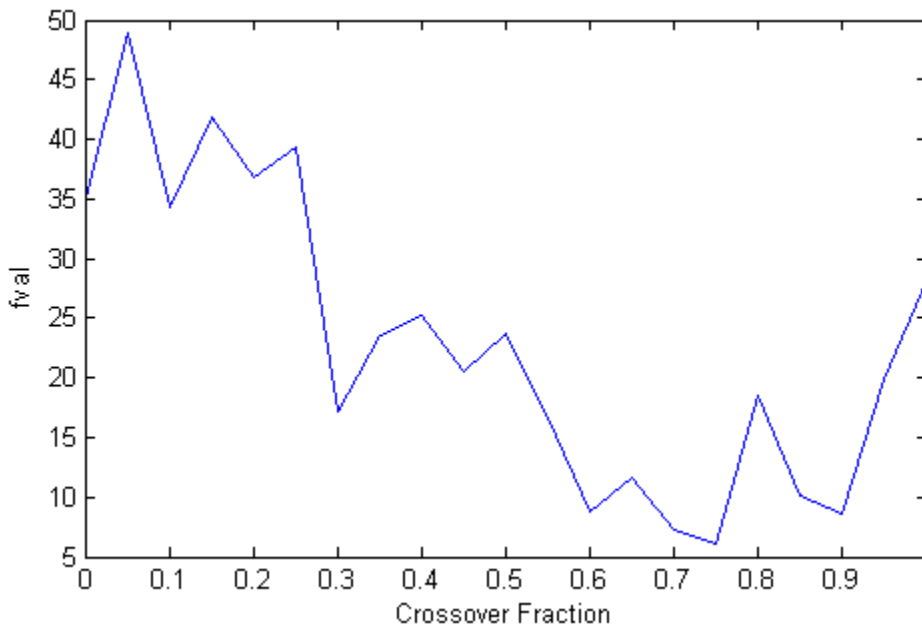
The plot suggests that you get the best results by setting `options.CrossoverFraction` to a value somewhere between `0.6` and `0.95`.

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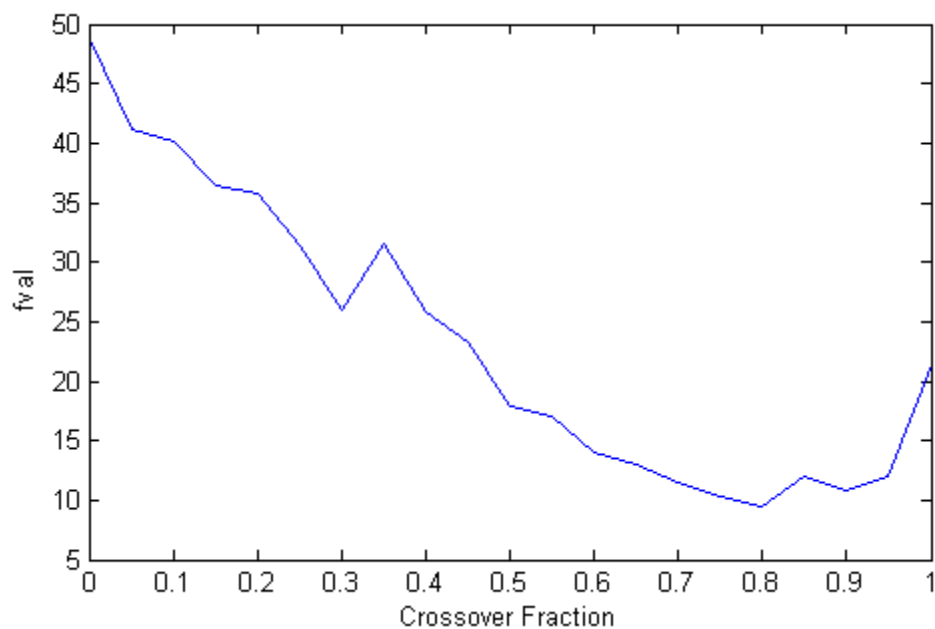
The plot narrows the range of best choices for `options.CrossoverFraction` to values between `0.7` and `0.9`.

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 start 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 9-29.

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 start 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 5-64 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, ga 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 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. Set upper and lower bounds for all generations in the **Bounds** fields in the **Constraints** panel.

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

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 5-8. The minimum value of the function is 0, which occurs at the origin.

To run the example, open the ga solver in the Optimization Tool by entering optimtool('ga') at the command line. Set the following:

- Set **Fitness function** to @rastriginsfcn.
- Set **Number of variables** to 2.
- Select **Best fitness** in the **Plot functions** pane of the **Options** pane.

- Select **Distance** in the **Plot functions** pane.

- Set **Initial range** in the **Population** pane of the **Options** pane to [1;1.1].

Click **Start** in **Run solver and view results**. Although the results of genetic algorithm computations are random, your results are similar to the following figure, with a best fitness function value of approximately 2.
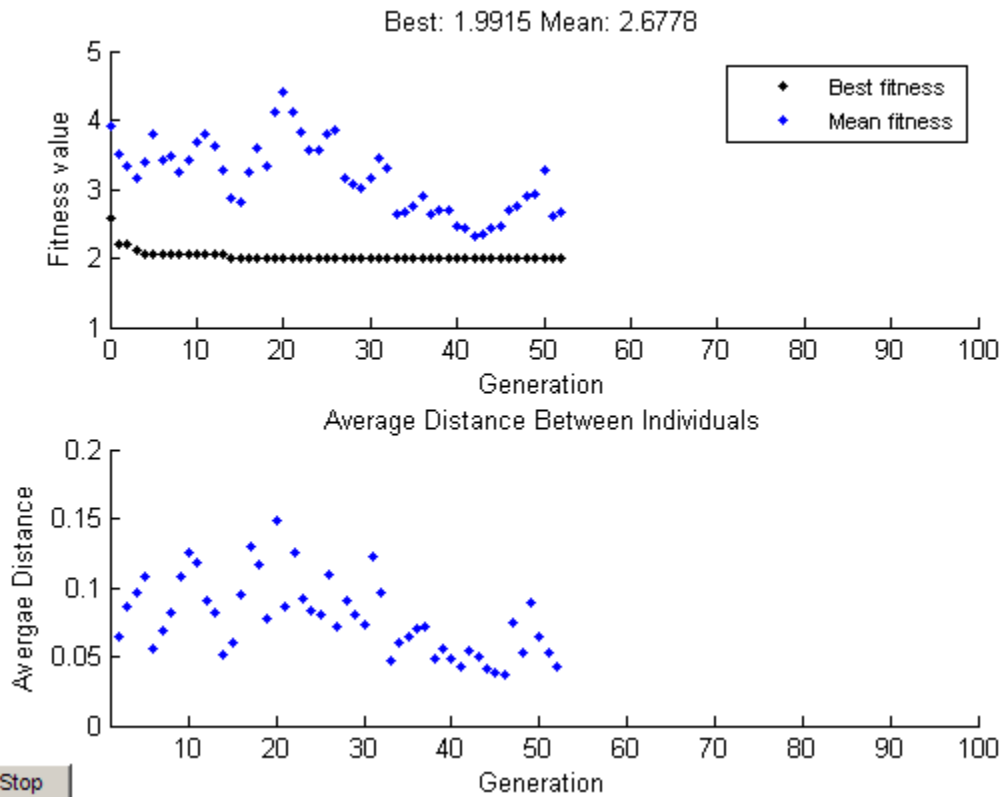
The upper plot, which displays the best fitness at each generation, shows little progress in lowering the fitness value. The lower plot shows the average distance between individuals at each generation, which is a good measure of the diversity of a population. For this setting of initial range, there is too little diversity for the algorithm to make progress.

Next, try setting **Initial range** to [1;100] and running the algorithm. This time the results are more variable. You might obtain a plot with a best fitness value of 35, as in the following plot. You might obtain different results.
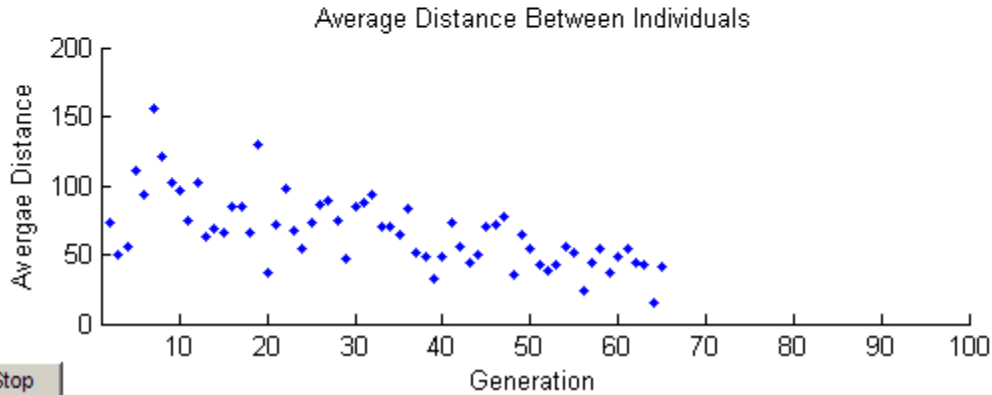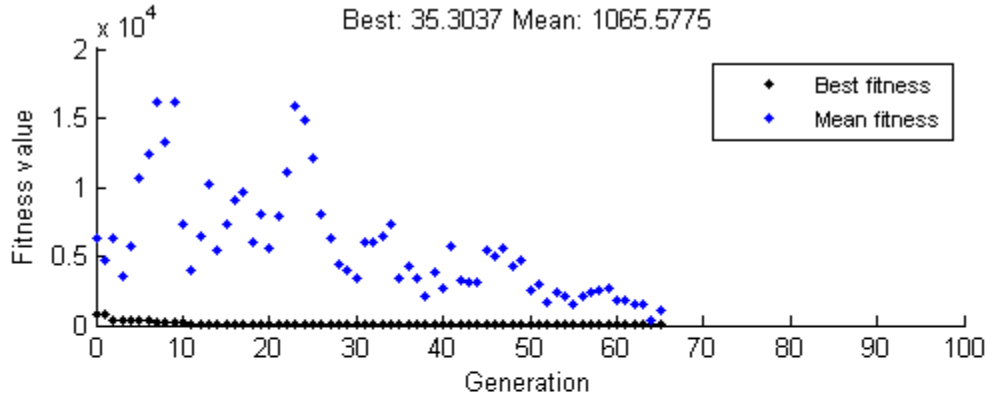
This time, the genetic algorithm makes progress, but because the average distance between individuals is so large, the best individuals are far from the optimal solution.

Finally, set **Initial range** to [1;2] and run the genetic algorithm. Again, there is variability in the result, but you might obtain a result similar to the following figure. Run the optimization several times, and you eventually obtain a final point near [0;0], with a fitness function value near 0.
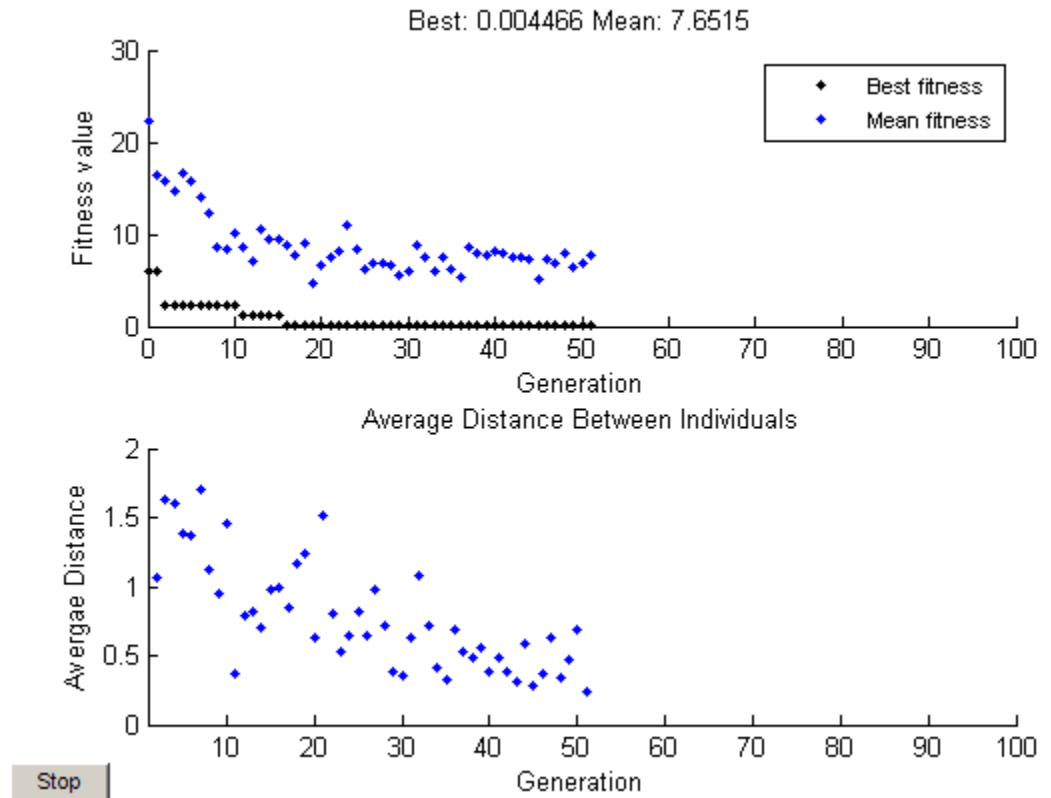
The diversity in this case is better suited to the problem, so ga usually returns a better result than in the previous two cases.

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:

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

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 % check to see if fitness function exists
    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) % if there is a fitness function
                    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 fitness function, and plots the population as it evolves. This plot function expects to have not only the usual inputs (`options,state,flag`), but also a function handle to the fitness function, `@lincontest6` in this example. To generate level curves, the custom plot function needs the fitness function.

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 `gaplotshowpopulation2`, and pass in `@lincontest6` as the fitness function handle:

```
options = gaoptimset('PlotFcn',...
          {{@gaplotshowpopulation2,@lincontest6}});
```

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:

Population plot in two dimension
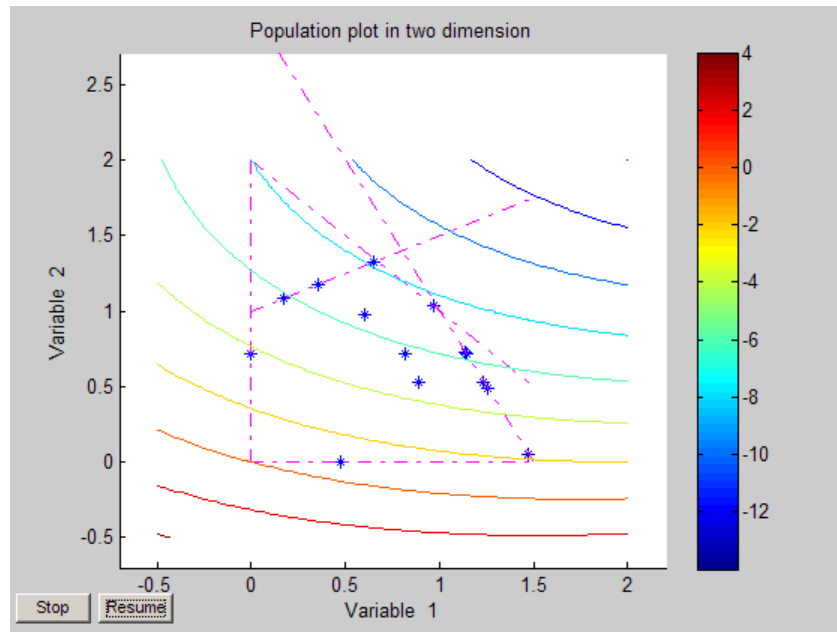
You can see that the initial population is biased to lie on the constraints.

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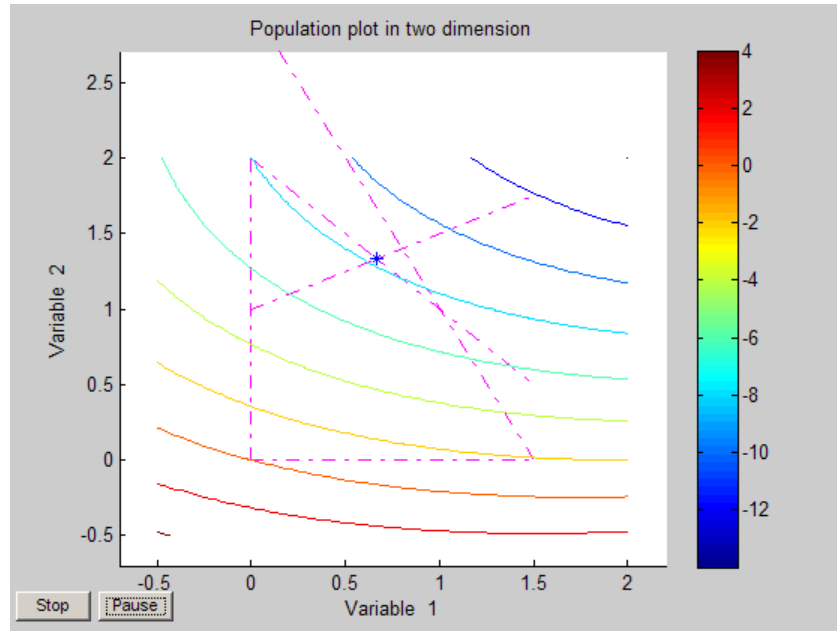
Setting the Population Size

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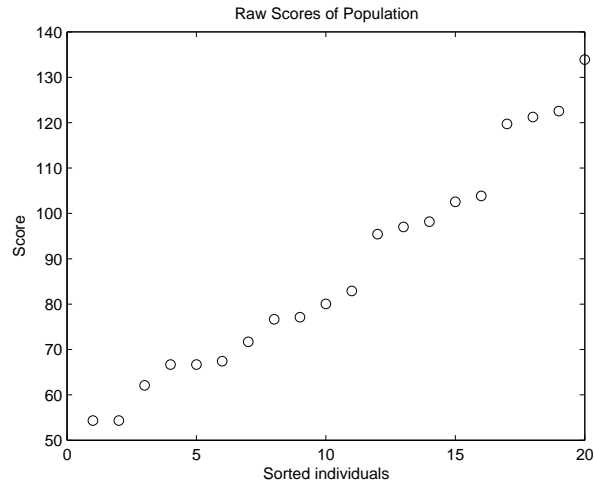
The following plot shows the scaled values of the raw scores using rank scaling.

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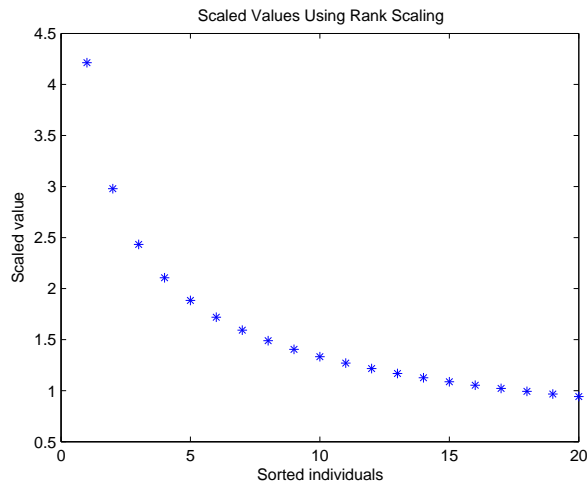
Because the algorithm minimizes the fitness function, lower raw scores have higher scaled values. Also, because rank scaling assigns values that depend

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
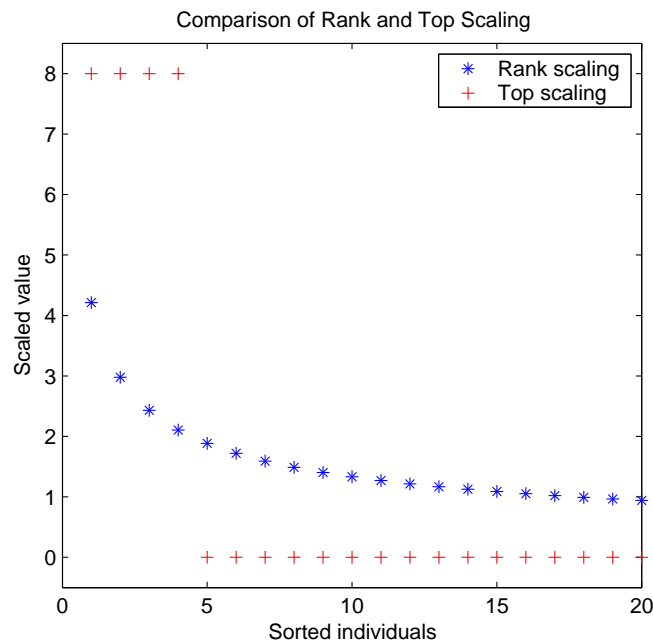
Because top scaling restricts parents to the fittest individuals, it creates less diverse populations than rank scaling. The following plot compares the

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

Variance of Distance Between Individuals Using Rank and Top Scaling
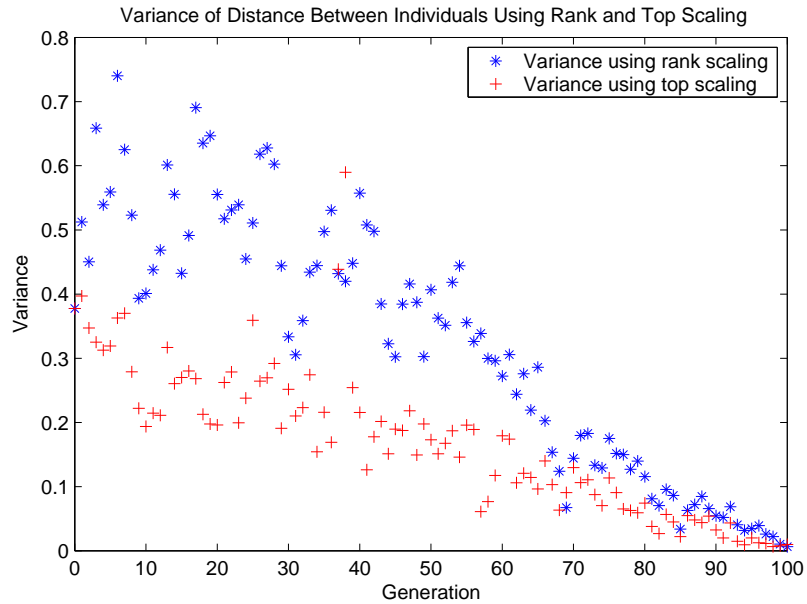
Selection

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 5-66 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 5-22 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 5-8. The following figure shows the plot.
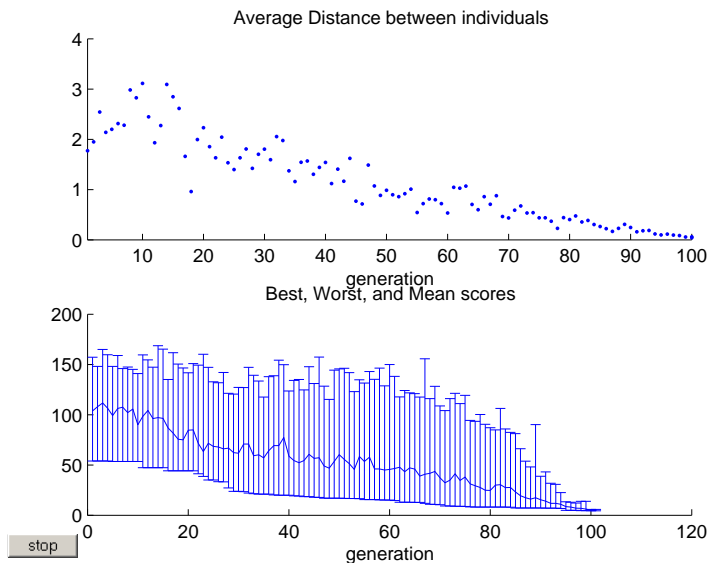
The upper plot displays the average distance between points in each generation. As the amount of mutation decreases, so does the average distance between individuals, which is approximately 0 at the final generation. The lower plot displays a vertical line at each generation, showing the range from the 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.
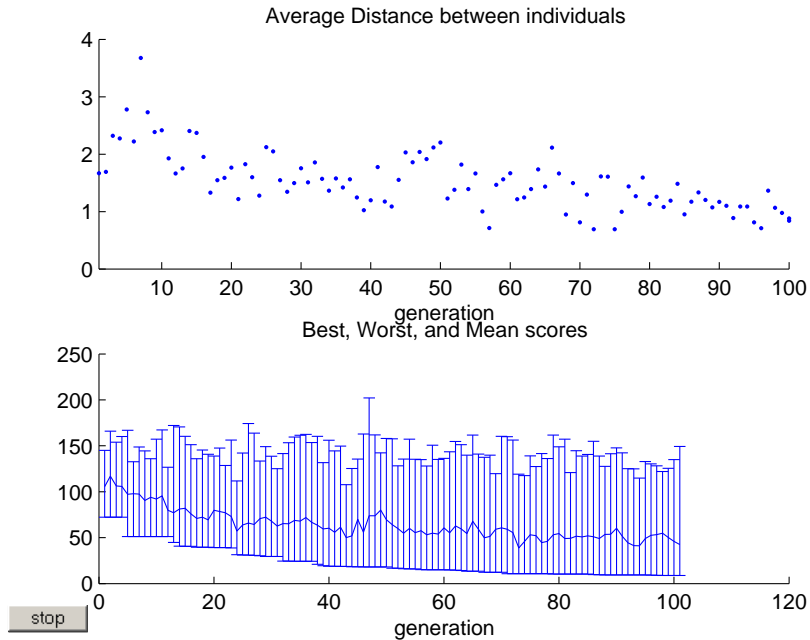
With **Shrink** set to 0.5, the average amount of mutation decreases by a factor of 1/2 by the final generation. As a result, the average distance between individuals decreases by approximately the same factor.

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,

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

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.
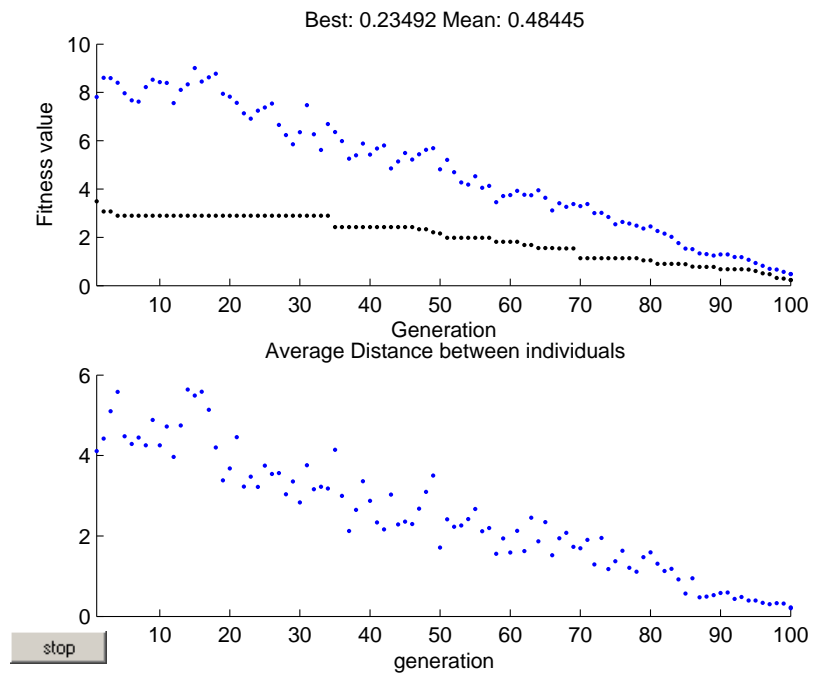
Crossover Without Mutation

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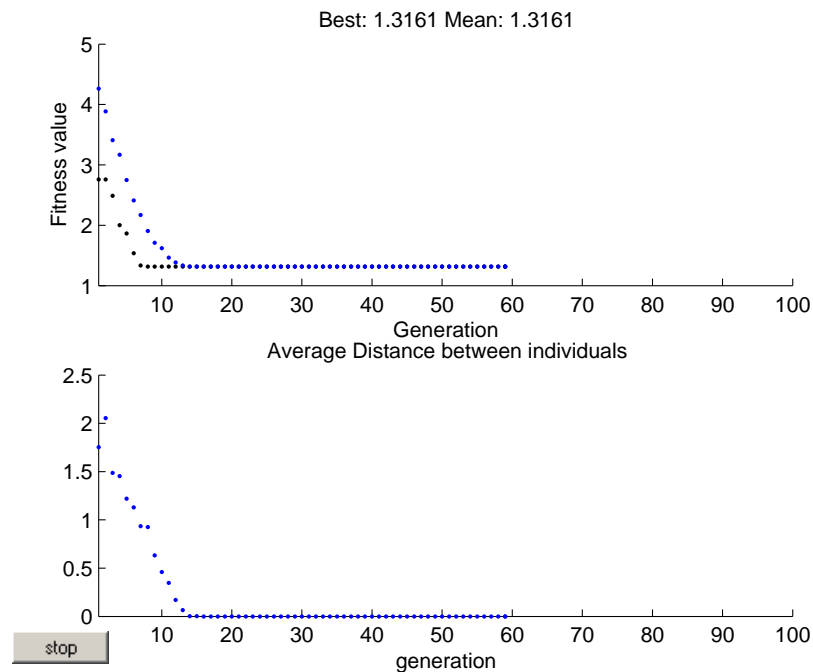
In this case, the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation. The algorithm generates the best individual that it can using these genes at generation number 8, where the best fitness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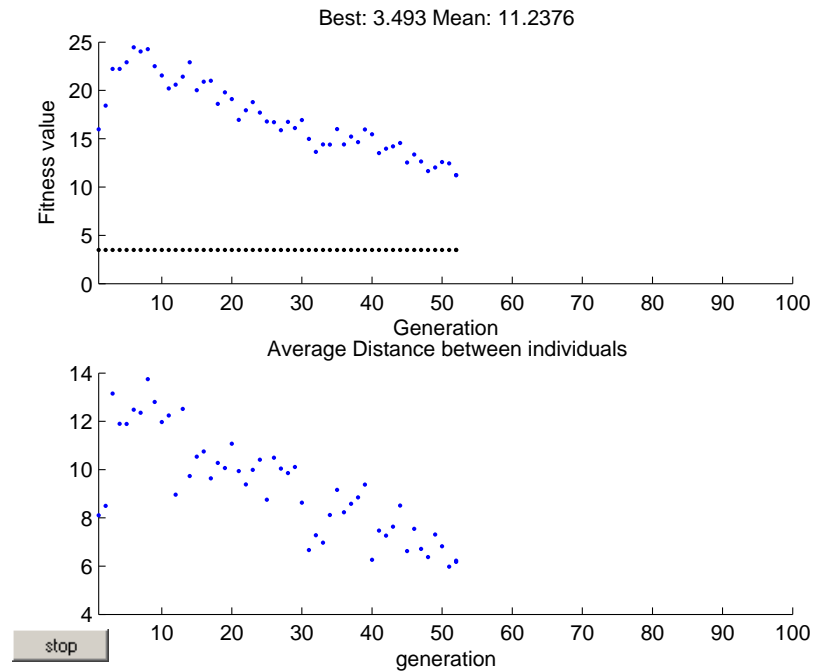
In this case, the random changes that the algorithm applies never improve the fitness value of the best individual at the first generation. While it improves the individual genes of other individuals, as you can see in the upper plot by the decrease in the mean value of the fitness function, these improved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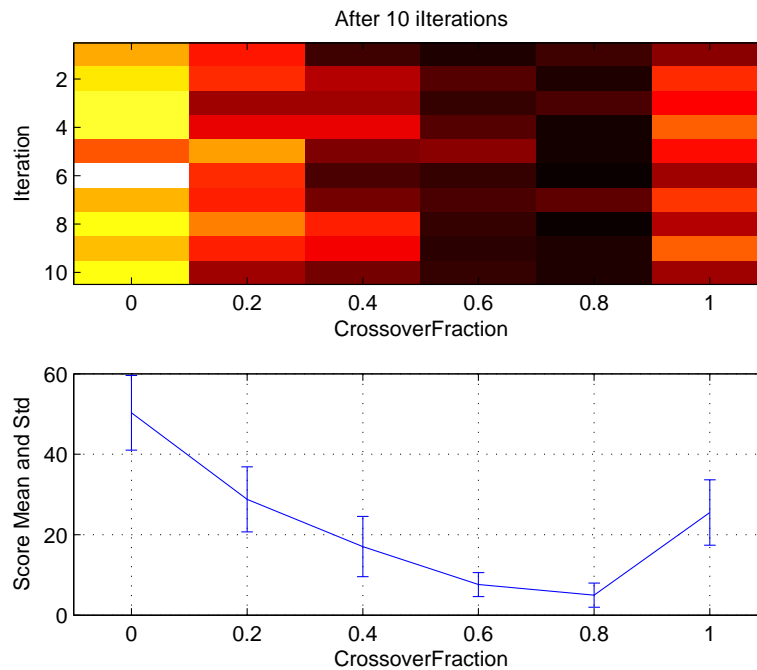
The lower plot shows the means and standard deviations of the best fitness values over 10 generations, for each of the values of the crossover fraction. The upper plot shows a color-coded display of the best fitness values in each generation.

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.

Example: Global vs. Local Minima with GA

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

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

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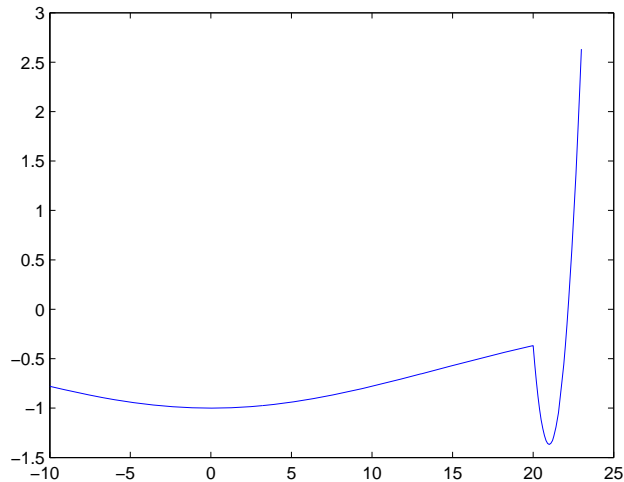
The function has two local minima, one at $x = 0$, where the function value is -1, and the other at $x = 21$, where the function value is $-1 - 1/e$. Since the latter value is smaller, the global minimum occurs at $x = 21$.

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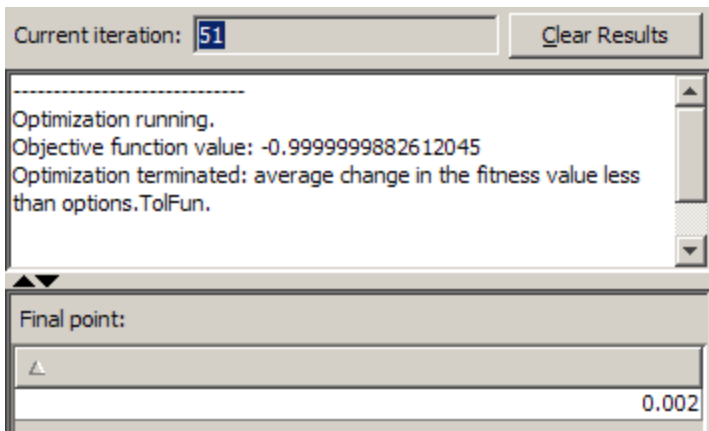
The following custom plot shows why the algorithm finds the local minimum rather than the global minimum. The plot shows the range of individuals in each generation and the best individual.
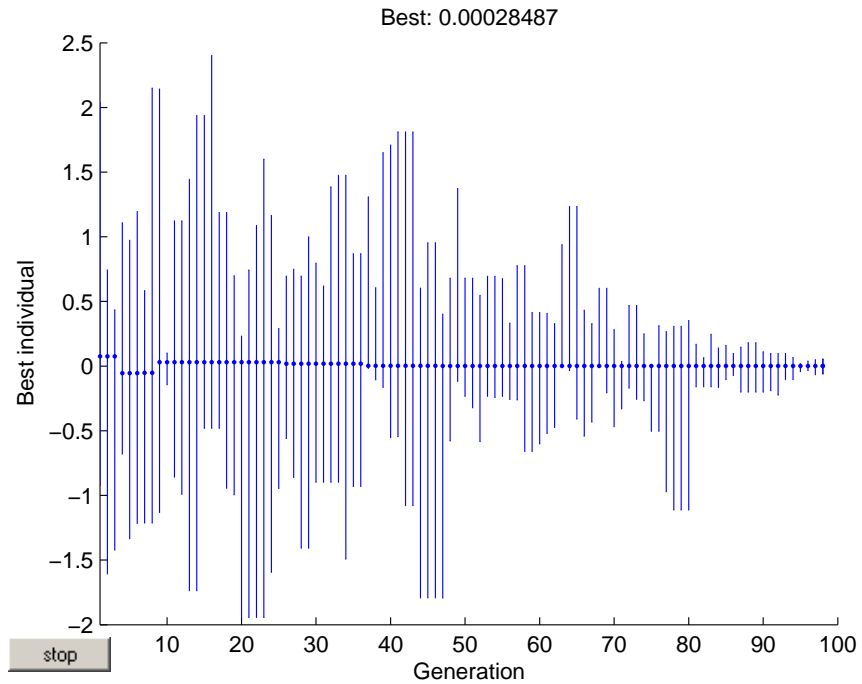
Note that all individuals are between -2 and 2.5. While this range is larger than the default **Initial range** of [0;1], due to mutation, it is not large enough to explore points near the global minimum at $x = 21$.

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.
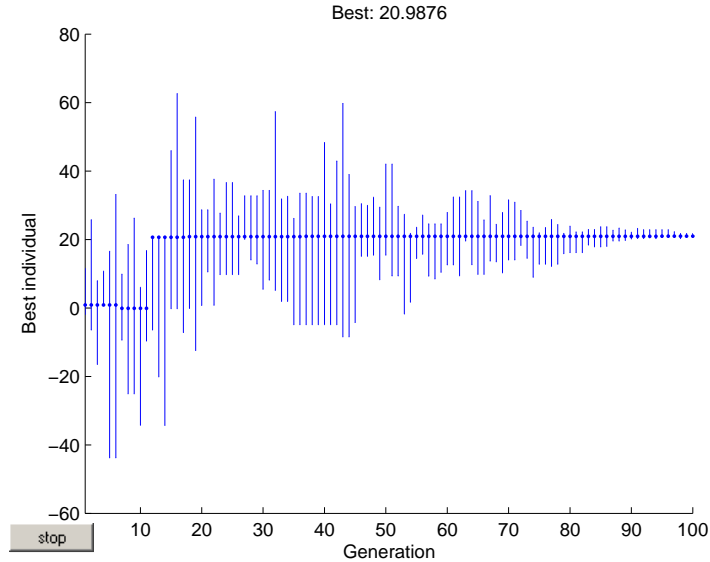
Using a Hybrid Function

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)
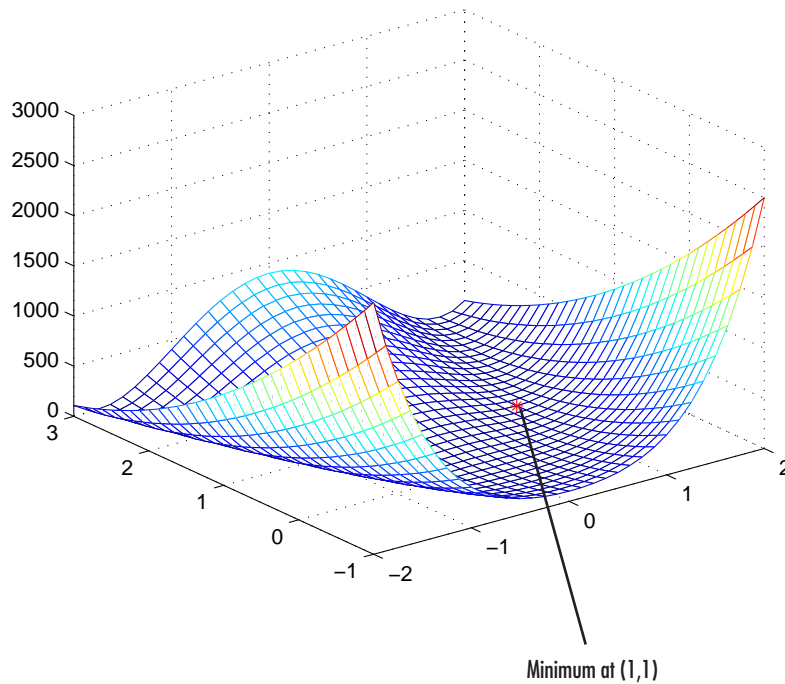
Global Optimization Toolbox software contains the `dejong2fcn.m` file, which computes Rosenbrock's function. To see a demo of this example, enter

```
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:
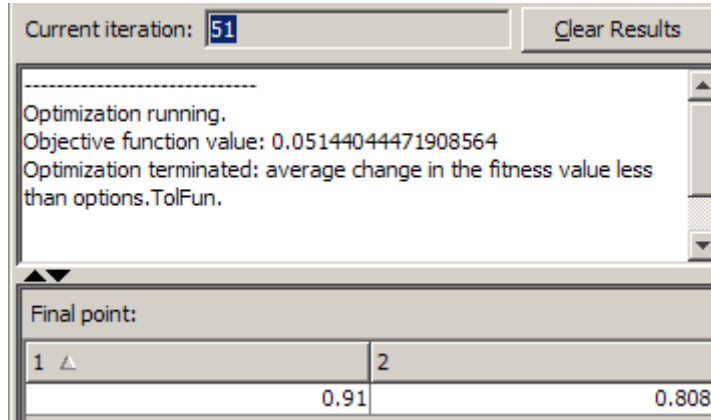
The final point is somewhat close to the true minimum at (1, 1). You can improve this result by setting **Hybrid function** to fminunc (in the **Hybrid function** options).
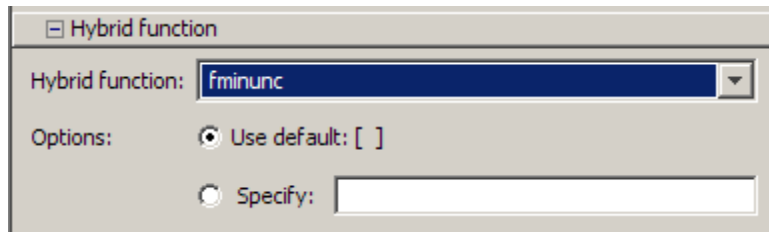
fminunc uses the final point of the genetic algorithm as its initial point. It returns a more accurate result, as shown in the **Run solver and view results** pane.
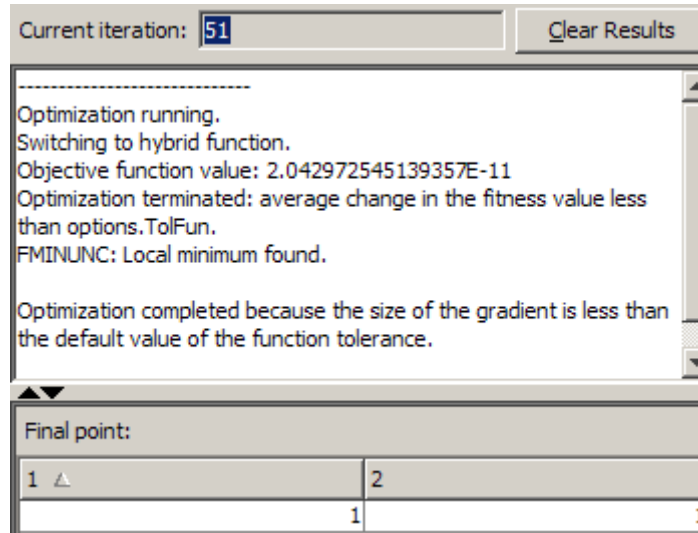
You can set options for the hybrid function separately from the calling function. Use `optimset` (or `psoptimset` for the `patternsearch` hybrid function) to create the options structure. For example:

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

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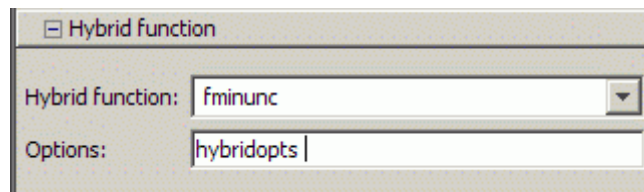
At the command line, the syntax is as follows:

```
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 5-24. 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.

Genetic algorithm stalls
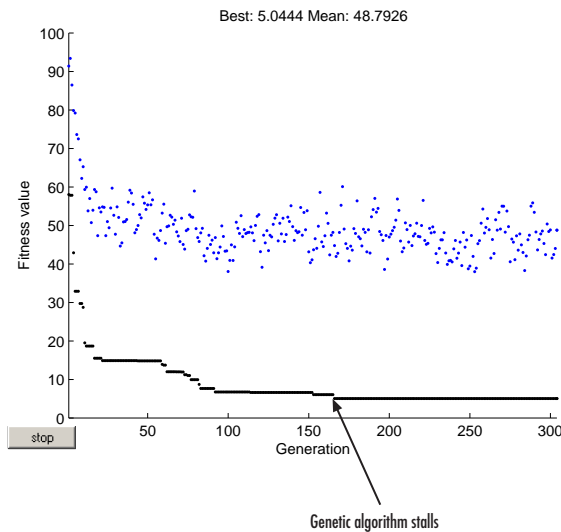
Note that the algorithm *stalls* at approximately generation number 170—that is, there is no immediate improvement in the fitness function after generation 170. If you restore **Stall generations** to its default value of 50, the algorithm would terminate at approximately generation number 230. If the 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 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

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

write the 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 **User function evaluation** pane, set the **Evaluate fitness and constraint functions** option to vectorized.

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

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

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

elapsed_time =

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

elapsed_time =

    0.5810
```

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

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

Begin by creating the fitness and constraint functions. First, create a 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 a 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)

Optimization terminated: average change in the fitness value
less than options.TolFun and constraint violation is
less than options.TolCon.

x =
```

```
       0.8122    12.3122


fval =
   1.3578e+004
```

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 mutation function `mutationguassian` does not necessarily obey the bound constraints. So when there are bound or linear constraints, the default `ga` mutation function is `mutationadaptfeasible`. 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.

To see the progress of the optimization, use the `gaoptimset` function to create an options structure that selects 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('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: average change in the fitness value
less than options.TolFun and constraint violation is
less than options.TolCon.

x =
    0.8122    12.3123


fval =
  1.3578e+004
```
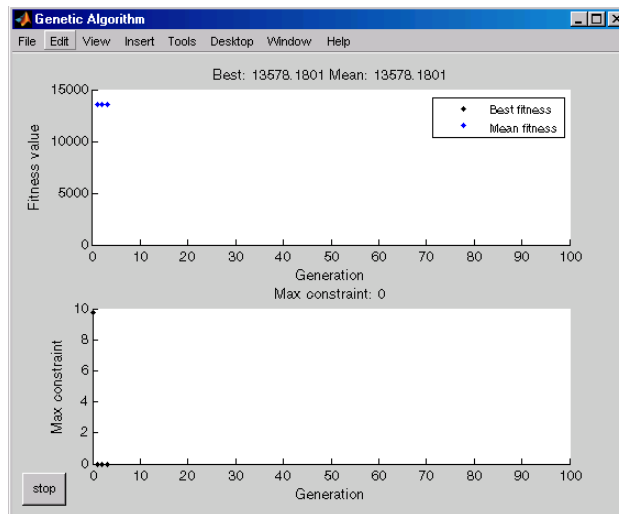

You can provide a start point for the minimization to the ga function by specifying the InitialPopulation option. The ga function will use the first individual from InitialPopulation as a start point for a constrained minimization.

```
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: average change in the fitness value
   less than options.TolFun and constraint violation is
   less than options.TolCon.

   x =
       0.8122   12.3122

   fval =
     1.3578e+004
```

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 4-72 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.

6

Using Simulated Annealing

What Is Simulated Annealing?

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.

Performing a Simulated Annealing Optimization

Calling simulannealbnd at the Command Line

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 a file.

"Using Simulated Annealing from the Command Line" on page 6-14 provides a detailed description of using the function `simulannealbnd` and creating the options structure.

Using the Optimization Tool

To open the Optimization Tool, enter

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

at the command line, or enter `optimtool` and then choose `simulannealbnd` from the **Solver** menu.
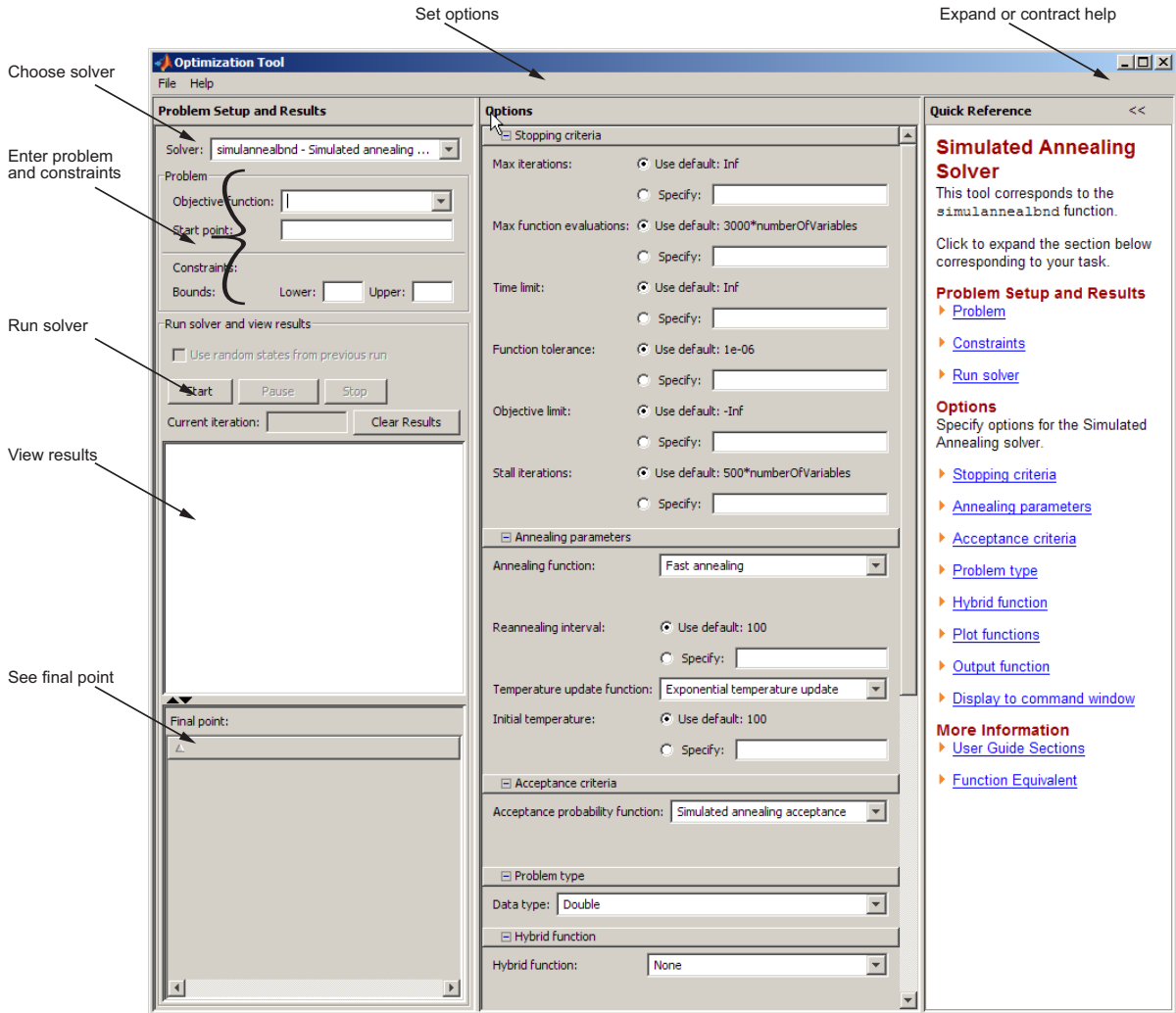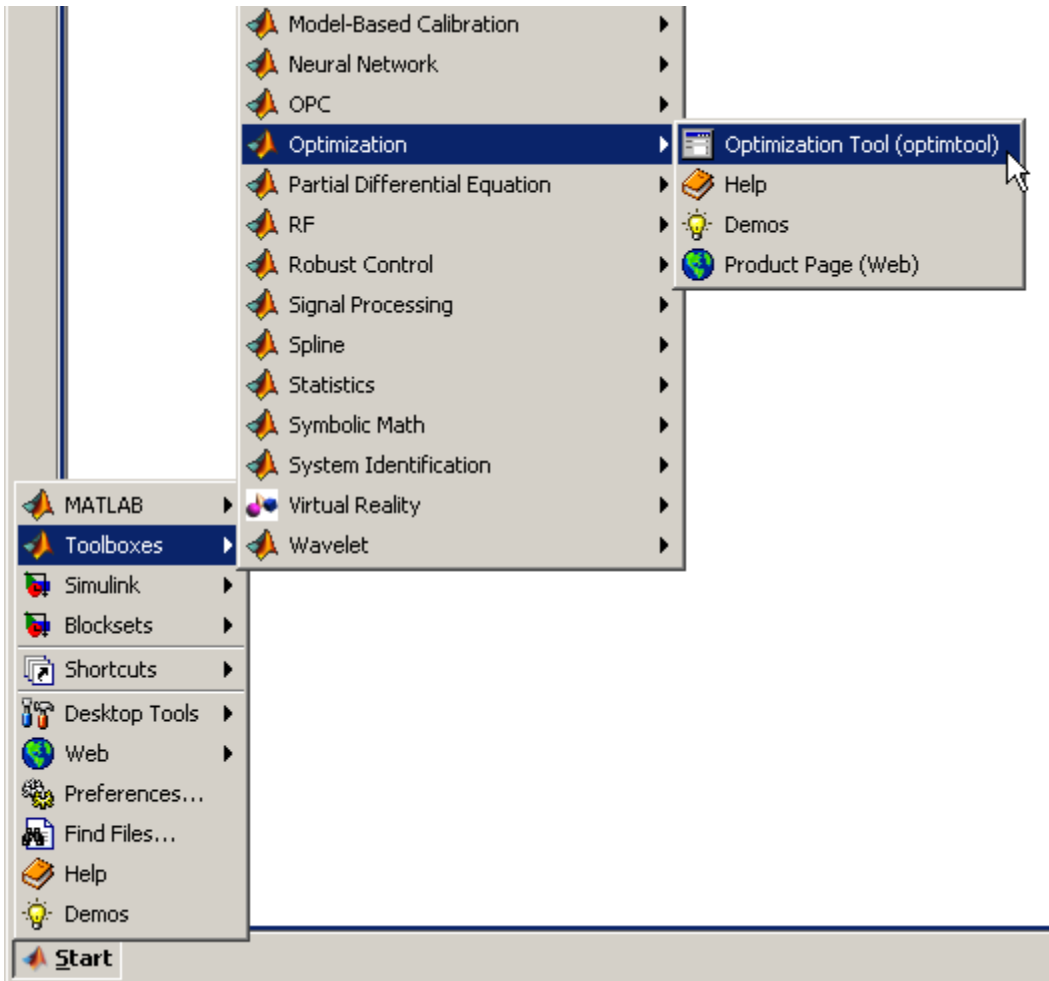
You can also start the tool from the MATLAB **Start** menu as pictured:

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 a file that computes the objective function. "Computing Objective Functions" on page 2-2 explains how write this 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 "Computing Objective Functions" on page 2-2, 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 documentation.
- See "Minimizing Using the Optimization Tool" on page 6-8 for an example of using the tool with the function simulannealbnd.

Example: Minimizing De Jong's Fifth Function

In this section...
"Description" on page 6-7
"Minimizing at the Command Line" on page 6-8
"Minimizing Using the Optimization Tool" on page 6-8

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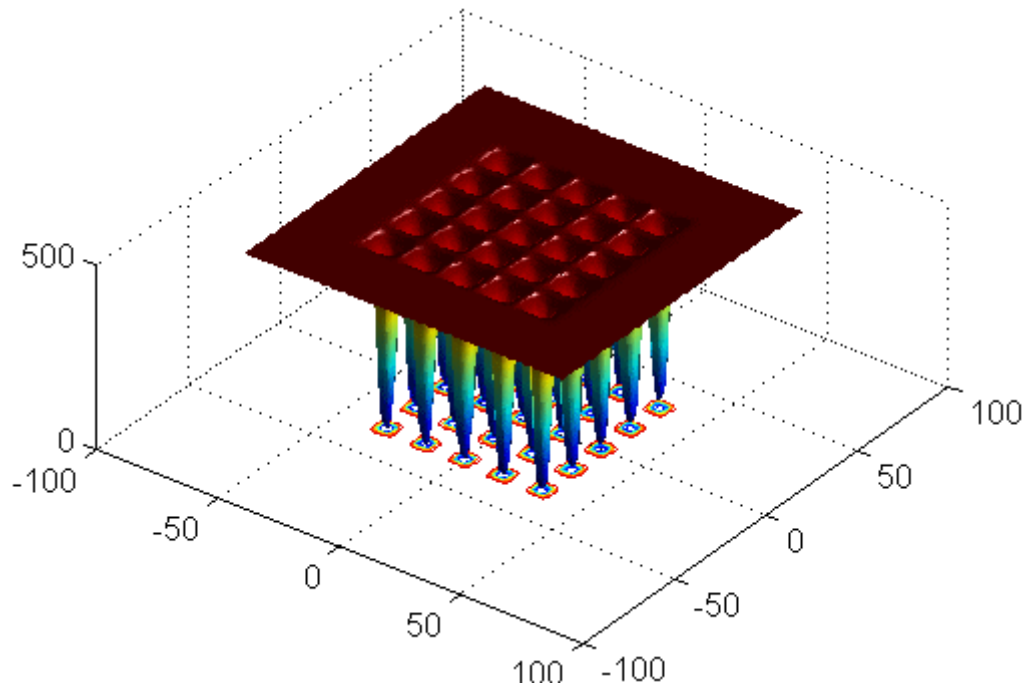
Many standard optimization algorithms get stuck in local minima. Because the simulated annealing algorithm performs a wide random search, the chance of being trapped in local minima is decreased.

Note Because simulated annealing uses random number generators, each time you run this algorithm you can get different results. See "Reproducing Your Results" on page 6-17 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: 1538   Clear Results

```
---------------------------
Optimization running.
Objective function value: 9.803897942861981
Optimization terminated: change in best function value less than
options.TolFun.
```

Final point:

1 △	2
31.943	-15.972

Some Simulated Annealing Terminology

In this section...
"Objective Function" on page 6-10
"Temperature" on page 6-10
"Annealing Schedule" on page 6-10
"Reannealing" on page 6-10

Objective Function

The *objective function* is the function you want to optimize. Global Optimization Toolbox algorithms attempt to find the minimum of the objective function. Write the objective function as a file or anonymous function, and pass it to the solver as a function 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 in 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 Works

In this section...
"Outline of the Algorithm" on page 6-12
"Stopping Conditions for the Algorithm" on page 6-12

Outline of the Algorithm

The following is an outline of the steps performed for the simulated annealing algorithm:

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. The algorithm accepts a worse point based on an acceptance probability.

3 The algorithm systematically lowers the temperature, 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 criterion is met. See "Stopping Conditions for the Algorithm" on page 6-12.

Stopping Conditions for the Algorithm

The simulated annealing algorithm uses 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`.

Using Simulated Annealing from the Command Line

Running simulannealbnd With the Default Options

To run the simulated annealing algorithm with the default options, call simulannealbnd with the syntax

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

The input arguments to simulannealbnd are

- @objfun — A function handle to the file that computes the objective function. "Computing Objective Functions" on page 2-2 explains how to write this 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.

You can run the example described in "Example: Minimizing De Jong's Fifth Function" on page 6-7 from the command line 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` 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` reference pages for more information about these arguments.

Setting Options for simulannealbnd at the Command Line

You can specify options by passing an options structure as an input argument to `simulannealbnd` 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 `saoptimset` function:

```
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'
```

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 algorithm is stochastic—that is, it makes random choices—you get slightly different results each time you run it. The algorithm uses the default MATLAB pseudorandom number stream. For more information about random number streams, see `RandStream`. Each time the algorithm calls the stream, its state changes. So the next time the algorithm calls the stream, it returns a different random number.

If you need to reproduce your results exactly, call `simulannealbnd` with the `output` argument. The `output` structure contains the current random number generator state in the `output.rngstate` field. Reset the state before running the function again.

For example, to reproduce the output of `simulannealbnd` 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 =
   31.9361  -31.9457

fval =
    4.9505
```

The state of the random number generator, `rngstate`, is stored in `output`:

```
output =
     iterations: 1754
      funccount: 1769
        message: 'Optimization terminated:...
   change in best function value less than options.TolFun.'
        rngstate: [1x1 struct]
    problemtype: 'unconstrained'
    temperature: [2x1 double]
       totaltime: 1.1094
```

Reset the stream by entering

```
stream = RandStream.getDefaultStream;
stream.State = output.rngstate.state;
```

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.
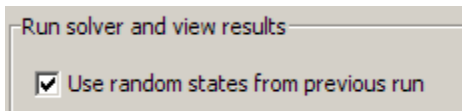
Note If you do not need to reproduce your results, it is better not to set the states of `RandStream`, so that you get the benefit of the randomness in these algorithms.

Simulated Annealing Examples

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

7

Multiobjective Optimization

- "What Is Multiobjective Optimization?" on page 7-2
- "Using gamultiobj" on page 7-5
- "References" on page 7-14

What Is Multiobjective Optimization?

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 7-3.
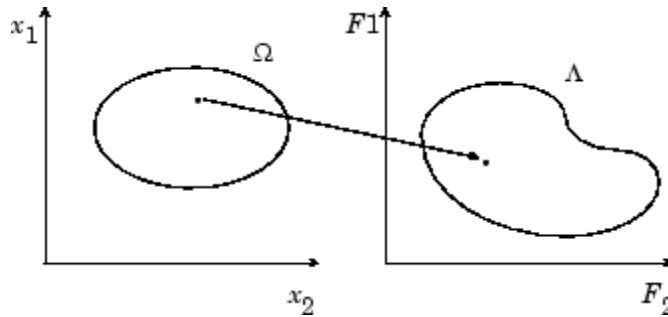
Figure 7-1: Mapping from Parameter Space into Objective Function Space

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

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

In the two-dimensional representation of the figure Set of Noninferior Solutions on page 7-4, the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.
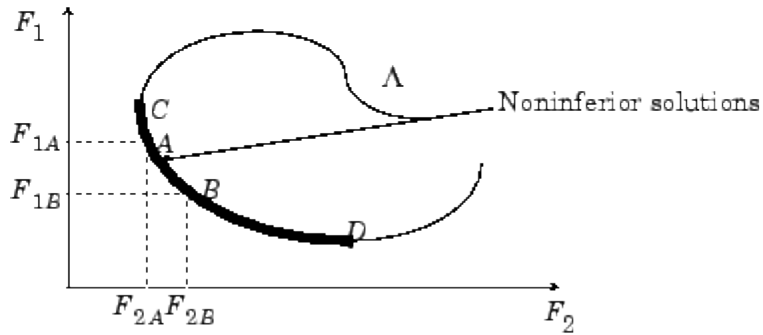
Figure 7-2: Set of Noninferior Solutions

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
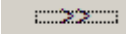
You can access gamultiobj from the Optimization Tool GUI. Enter

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

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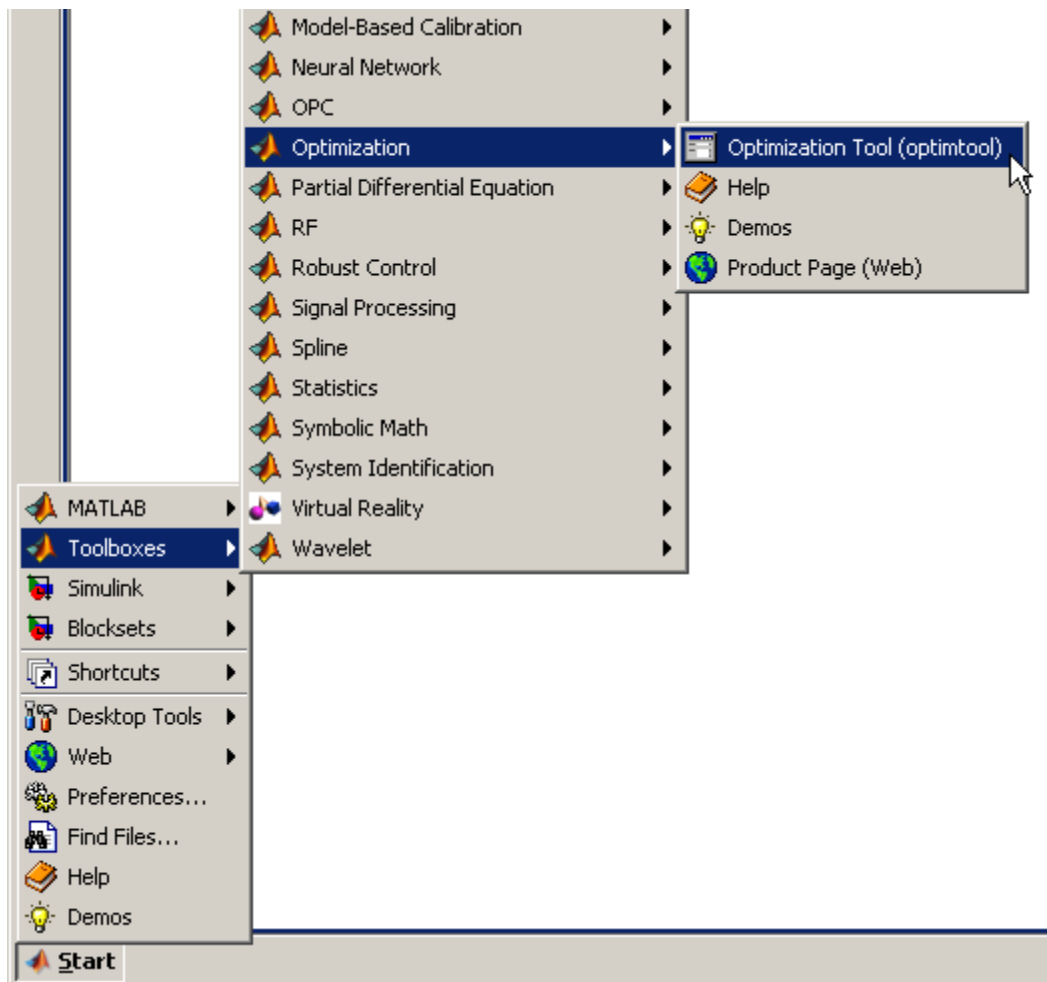
If the **Quick Reference** help pane is closed, you can open it by clicking
the ">>" button on the upper right of the GUI: [≫]. All the options
available are explained briefly in the help pane.

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:
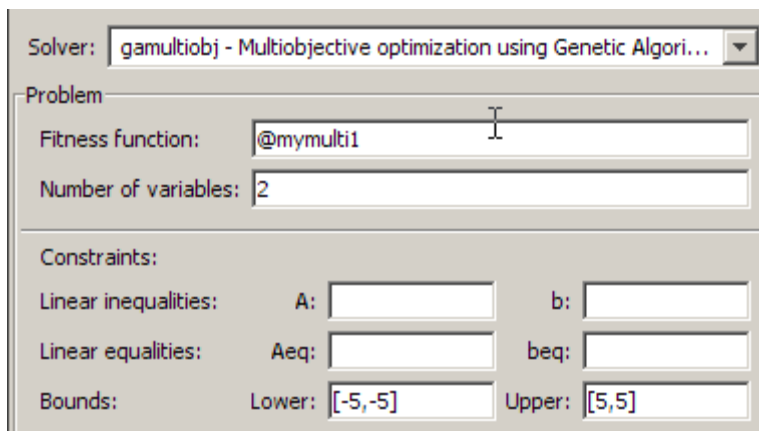
```
function f = mymulti1(x)

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 this function file before proceeding.

Performing the Optimization with Optimization Tool

1 To define the optimization problem, start 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.
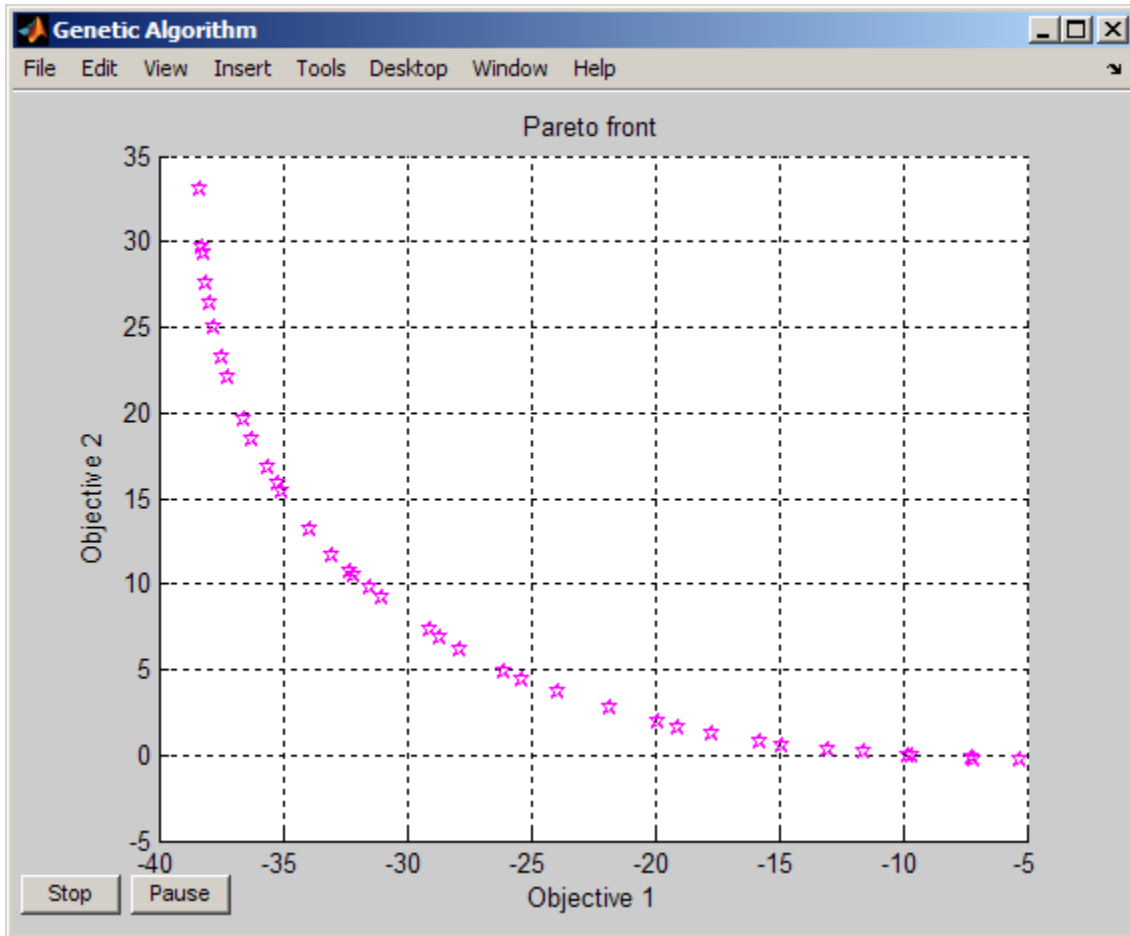
This plot shows the tradeoff between the two components of *f*. It is plotted in objective function space; see the figure Set of Noninferior Solutions on page 7-4.

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: 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	-5.252	-0.25	-0.707	0.707
2	-38.333	33.035	2.671	-1.976
3	-23.319	3.331	1.632	-1.29
4	-25.192	4.564	1.725	-1.205
5	-24.433	4.063	1.688	-1.201
6	-38.029	27.033	2.551	-1.912
7	-35.82	17.663	2.313	-1.825
8	-5.252	-0.25	-0.707	0.707
9	-15.657	0.822	1.284	-0.951
10	-38.139	28.494	2.581	-1.955
11	-34.906	15.547	2.246	-1.795
12	-37.756	25.435	2.514	-1.939
13	-30.779	8.888	1.992	-1.547
14	-31.898	10.533	2.06	-1.68
15	-36.18	18.652	2.342	-1.839
16	-18.17	1.477	1.402	-1.013
17	-35.571	16.653	2.285	-1.69
18	-16.794	1.103	1.338	-0.973

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	-38.333	33.035	2.671	-1.976
23	-38.317	31.713	2.646	-1.973
29	-38.184	29.338	2.598	-1.976
10	-38.139	28.494	2.581	-1.955
6	-38.029	27.033	2.551	-1.912
12	-37.756	25.435	2.514	-1.939
41	-37.566	23.748	2.476	-1.823
31	-37.231	22.031	2.434	-1.814
33	-36.876	20.601	2.397	-1.813
24	-36.532	19.477	2.367	-1.724
15	-36.18	18.652	2.342	-1.839
7	-35.82	17.663	2.313	-1.825
17	-35.571	16.653	2.285	-1.69
11	-34.906	15.547	2.246	-1.795
30	-34.396	14.905	2.22	-1.82
25	-34.302	13.968	2.197	-1.701
19	-33.521	12.609	2.148	-1.573
34	-33.071	12.232	2.128	-1.727

Pareto front - function values and decision variables

Index	f1 ▽	f2	x1	x2
1	-5.252	-0.25	-0.707	0.707
8	-5.252	-0.25	-0.707	0.707
20	-5.702	-0.248	-0.738	0.726
21	-7.922	-0.187	0.88	-0.848
36	-8.209	-0.128	0.899	-0.937
28	-9.95	-0.032	0.996	-0.789
32	-11.649	0.148	1.086	-0.836
42	-12.99	0.398	1.157	-0.818
9	-15.657	0.822	1.284	-0.951
18	-16.794	1.103	1.338	-0.973
16	-18.17	1.477	1.402	-1.013
22	-19.399	1.77	1.455	-1.219
26	-20.13	1.998	1.488	-1.186
37	-20.844	2.26	1.52	-1.207
39	-22.002	2.936	1.579	-1.389
3	-23.319	3.331	1.632	-1.29
5	-24.433	4.063	1.688	-1.201
4	-25.192	4.564	1.725	-1.205

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 7-3.
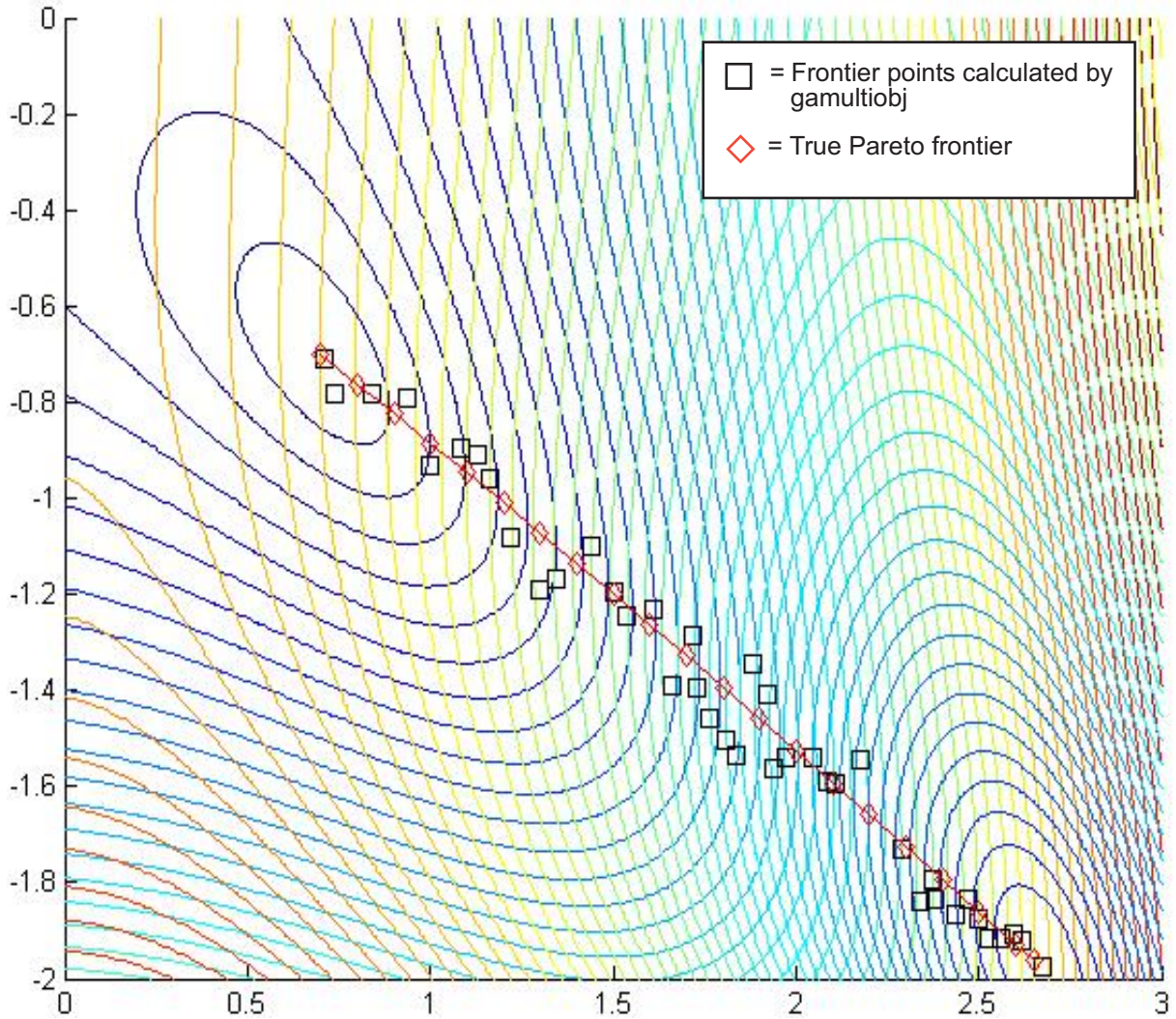
Contours of objective functions, and Pareto frontier

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.

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.

8

Parallel Processing

Background

Types of Parallel Processing in Global Optimization Toolbox

Parallel processing is an attractive way to speed optimization algorithms. To use parallel processing, you must have a Parallel Computing Toolbox license, and have a MATLAB worker pool open (`matlabpool`). For more information, see "How to Use Parallel Processing" on page 8-11.

Global Optimization Toolbox solvers use parallel computing in various ways.

Solver	Parallel?	Parallel Characteristics
`GlobalSearch`	×	No parallel functionality. However, `fmincon` can use parallel gradient estimation when run in `GlobalSearch`. See "Using Parallel Computing with fmincon, fgoalattain, and fminimax" in the Optimization Toolbox documentation.
`MultiStart`	✓	Start points distributed to multiple processors. From these points, local solvers run to completion. For more details, see "MultiStart" on page 8-5 and "How to Use Parallel Processing" on page 8-11.
		For `fmincon`, no parallel gradient estimation with parallel `MultiStart`.
`ga`, `gamultiobj`	✓	Population evaluated in parallel, which occurs once per iteration. For more details, see "Genetic Algorithm" on page 8-8 and "How to Use Parallel Processing" on page 8-11.
		No vectorization of fitness or constraint functions.
`patternsearch`	✓	Poll points evaluated in parallel, which occurs once per iteration. For more details, see "Pattern Search" on page 8-7 and "How to Use Parallel Processing" on page 8-11.
		No vectorization of fitness or constraint functions.
`simulannealbnd`	×	No parallel functionality. However, `simulannealbnd` can use a hybrid function that runs in parallel. See "Simulated Annealing" on page 8-10.

In addition, several solvers have hybrid functions that run after they finish. Some hybrid functions can run in parallel. Also, most `patternsearch` search methods can run in parallel. For more information, see "Parallel Search Functions or Hybrid Functions" on page 8-14.

How Toolbox Functions Distribute Processes

- "parfor Characteristics and Caveats" on page 8-3
- "MultiStart" on page 8-5
- "GlobalSearch" on page 8-6
- "Pattern Search" on page 8-7
- "Genetic Algorithm" on page 8-8
- "Parallel Computing with gamultiobj" on page 8-9
- "Simulated Annealing" on page 8-10

parfor Characteristics and Caveats

No Nested parfor Loops. Solvers employ the Parallel Computing Toolbox `parfor` function to perform parallel computations.

Note `parfor` does not work in parallel when called from within another `parfor` loop.

Suppose, for example, your objective function `userfcn` calls `parfor`, and you want to call `fmincon` using `MultiStart` and parallel processing. Suppose also that the conditions for parallel gradient evaluation of `fmincon` are satisfied, as given in "Parallel Optimization Functionality". The figure When parfor Runs In Parallel on page 8-4 shows three cases:

1 The outermost loop is parallel `MultiStart`. Only that loop runs in parallel.

2 The outermost `parfor` loop is in `fmincon`. Only `fmincon` runs in parallel.

3 The outermost parfor loop is in userfcn. In this case, userfcn can use parfor in parallel.

Bold indicates the function that runs in parallel

①
```
...
problem = createOptimProblem(fmincon,'objective',@userfcn,...)
ms = MultiStart('UseParallel','always');
x = run(ms,problem,10)
...
```
Only the outermost parfor loop runs in parallel

If fmincon UseParallel option = 'always' fmincon estimates gradients in parallel

②
```
...
x = fmincon(@userfcn,...)
...
```

If fmincon UseParallel option = 'never' userfcn can use parfor in parallel

③
```
...
x = fmincon(@userfcn,...)
...
```

When parfor Runs In Parallel

Parallel Random Numbers Are Not Reproducible. Random number sequences in MATLAB are pseudorandom, determined from a *seed*, or an initial setting. Parallel computations use seeds that are not necessarily controllable or reproducible. For example, each instance of MATLAB has a default global setting that determines the current seed for random sequences.

For patternsearch, if you select MADS as a poll or search method, parallel pattern search does not have reproducible runs. If you select the genetic algorithm or Latin hypercube as search methods, parallel pattern search does not have reproducible runs.

For ga and gamultiobj, parallel population generation gives nonreproducible results.

MultiStart is different. You *can* have reproducible runs from parallel MultiStart. Runs are reproducible because MultiStart generates pseudorandom start points locally, and then distributes the start points to parallel processors. Therefore, the parallel processors do not use random

numbers. For more details, see "Parallel Processing and Random Number Streams" on page 3-61.

Limitations and Performance Considerations. More caveats related to `parfor` appear in the "Limitations" section of the Parallel Computing Toolbox documentation.

For information on factors that affect the speed of parallel computations, and factors that affect the results of parallel computations, see "Improving Performance with Parallel Computing" in the Optimization Toolbox documentation. The same considerations apply to parallel computing with Global Optimization Toolbox functions.

MultiStart

`MultiStart` can automatically distribute a problem and start points to multiple processes or processors. The problems run independently, and `MultiStart` combines the distinct local minima into a vector of `GlobalOptimSolution` objects. `MultiStart` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.
- Enable parallel computing with `matlabpool`, a Parallel Computing Toolbox function.
- Set the `UseParallel` property to `'always'` in the `MultiStart` object:

```
ms = MultiStart('UseParallel','always');
```

When these conditions hold, `MultiStart` distributes a problem and start points to processes or processors one at a time. The algorithm halts when it reaches a stopping condition or runs out of start points to distribute. If the `MultiStart Display` property is `'iter'`, then `MultiStart` displays:

```
Running the local solvers in parallel.
```

For an example of parallel `MultiStart`, see "Example: Parallel MultiStart" on page 3-74.

Implementation Issues in Parallel MultiStart. fmincon cannot estimate gradients in parallel when used with parallel MultiStart. This lack of parallel gradient estimation is due to the limitation of parfor described in "No Nested parfor Loops" on page 8-3.

fmincon can take longer to estimate gradients in parallel rather than in serial. In this case, using MultiStart with parallel gradient estimation in fmincon amplifies the slowdown. For example, suppose the ms MultiStart object has UseParallel set to 'never'. Suppose fmincon takes 1 s longer to solve problem with problem.options.UseParallel set to 'always'. Then run(ms,problem,200) takes 200 s longer than the same run with problem.options.UseParallel set to 'never'

Note When executing serially, parfor loops run slower than for loops. Therefore, for best performance, set your local solver UseParallel option to 'never' when the MultiStart UseParallel property is 'always'.

GlobalSearch

GlobalSearch does not distribute a problem and start points to multiple processes or processors. However, when GlobalSearch runs the fmincon local solver, fmincon can estimate gradients by parallel finite differences. fmincon uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.

- Enable parallel computing with matlabpool, a Parallel Computing Toolbox function.

- Set the UseParallel option to 'always' with optimset. Set this option in the problem structure:

```
opts = optimset('UseParallel','always','Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',@myobj,...
    'x0',startpt,'options',opts);
```

For more details, see "Using Parallel Computing with fmincon, fgoalattain, and fminimax" in the Optimization Toolbox documentation.

Pattern Search

`patternsearch` can automatically distribute the evaluation of objective and constraint functions associated with the points in a pattern to multiple processes or processors. `patternsearch` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.

- Enable parallel computing with `matlabpool`, a Parallel Computing Toolbox function.

- Set the following options using `psoptimset` or the Optimization Tool:

  - `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. Furthermore, `patternsearch` overrides the setting of the `Cache` option, and uses the default `'off'` setting.

Parallel Search Function. `patternsearch` can optionally call a search function at each iteration. The search is parallel when you:

- Set `CompleteSearch` to `'on'`.

- Do not set the search method to `@searchneldermead` or `custom`.

- Set the search method to a `patternsearch` poll method or Latin hypercube search, and set `UseParallel` to `'always'`.

- Or, if you set the search method to `ga`, create a search method option structure with `UseParallel` set to `'always'`.

Implementation Issues in Parallel Pattern Search. The limitations on `patternsearch` options, listed in "Pattern Search" on page 8-7, arise partly from the limitations of `parfor`, and partly from the nature of parallel processing:

- `Cache` is overridden to be `'off'` — `patternsearch` implements `Cache` as a persistent variable. `parfor` does not handle persistent variables, because the variable could have different settings at different processors.

- `CompletePoll` is `'on'` — `CompletePoll` determines whether a poll stops as soon as `patternsearch` finds a better point. When searching in parallel, `parfor` schedules all evaluations simultaneously, and `patternsearch` continues after all evaluations complete. `patternsearch` cannot halt evaluations after they start.

- `Vectorized` is `'off'` — `Vectorized` determines whether `patternsearch` evaluates all points in a pattern with one function call in a vectorized fashion. If `Vectorized` is `'on'`, `patternsearch` does not distribute the evaluation of the function, so does not use `parfor`.

Genetic Algorithm

`ga` and `gamultiobj` can automatically distribute the evaluation of objective and nonlinear constraint functions associated with a population to multiple processors. `ga` uses parallel computing when you:

- Have a license for Parallel Computing Toolbox software.

- Enable parallel computing with `matlabpool`, a Parallel Computing Toolbox function.

- Set the following options using `gaoptimset` or the Optimization Tool:

  - `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.

Implementation Issues in Parallel Genetic Algorithm. The limitations on options, listed in "Genetic Algorithm" on page 8-8, arise partly from limitations of `parfor`, and partly from the nature of parallel processing:

- `Vectorized` is `'off'` — `Vectorized` determines whether `ga` evaluates an entire population with one function call in a vectorized fashion. If `Vectorized` is `'on'`, `ga` does not distribute the evaluation of the function, so does not use `parfor`.

`ga` can have a hybrid function that runs after it finishes; see "Using a Hybrid Function" on page 5-76. If you want the hybrid function to take advantage of parallel computation, set its options separately so that `UseParallel` is

'always'. If the hybrid function is patternsearch, set CompletePoll to 'on' so that patternsearch runs in parallel.

If the hybrid function is fmincon, set the following options with optimset to have parallel gradient estimation:

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

To find out how to write options for the hybrid function, see "Parallel Hybrid Functions" on page 8-15.

Parallel Computing with gamultiobj

Parallel computing with gamultiobj works almost the same as with ga. For detailed information, see "Genetic Algorithm" on page 8-8.

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 optionally runs after gamultiobj finishes its run. Each individual in the calculated Pareto frontier, that is, the final population found by gamultiobj, becomes the starting point for an optimization using fgoalattain. These optimizations run 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, set its options 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 9-47.

gamultiobj calls fgoalattain using a parfor loop, so fgoalattain does not estimate gradients in parallel when used as a hybrid function with gamultiobj. For more information, see "No Nested parfor Loops" on page 8-3.

Simulated Annealing

`simulannealbnd` does not run in parallel automatically. However, it can call hybrid functions that take advantage of parallel computing. To find out how to write options for the hybrid function, see "Parallel Hybrid Functions" on page 8-15.

How to Use Parallel Processing

In this section...
"Multicore Processors" on page 8-11
"Processor Network" on page 8-12
"Parallel Search Functions or Hybrid Functions" on page 8-14

Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a `matlabpool` of several parallel workers with a Parallel Computing Toolbox license. For a description of Parallel Computing Toolbox software, and the maximum number of parallel workers, see "Product Overview".

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

- Enter

  ```
  matlabpool open 2
  ```

  at the command line. The 2 specifies the number of MATLAB processes to start.

- Set your solver to use parallel processing.

MultiStart	Patternsearch	GA

`ms = MultiStart('UseParallel', 'always');`
 or
 `ms.UseParallel = 'always'`
	`options = psoptimset('UseParallel', 'always', 'CompletePoll', 'on', 'Vectorized', 'off');`	`options = gaoptimset('UseParallel', 'always', 'Vectorized', 'off');`

	For Optimization Tool: - **Options > User function evaluation > Evaluate**	For Optimization Tool: - **Options > User function evaluation > Evaluate**

MultiStart	Patternsearch	GA
	objective and constraint functions > in parallel - Options > Complete poll > on	fitness and constraint functions > in parallel

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

To stop computing optimizations in parallel, set `UseParallel` to `'never'`, or set the Optimization Tool not to compute in parallel. To halt all parallel computation, enter

```
matlabpool close
```

Processor 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 Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation, or the Administrator Guide documentation for MATLAB Distributed Computing Server.

To perform a basic check:

a 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 want to use.

b 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 files.

c Check whether a file is on the path of every worker by entering

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

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

```
filename not found.
```

2 Set your solver to use parallel processing.

MultiStart	Patternsearch	GA
ms = MultiStart('UseParallel', 'always'); or ms.UseParallel = 'always'	options = psoptimset('UseParallel', 'always', 'CompletePoll', 'on', 'Vectorized', 'off');	options = gaoptimset('UseParallel', 'always', 'Vectorized', 'off');
	For Optimization Tool: • **Options > User function evaluation > Evaluate objective and constraint functions > in parallel** • **Options > Complete poll > on**	For Optimization Tool: • **Options > User function evaluation > Evaluate fitness and constraint functions > in parallel**

After you establish your parallel computing environment, applicable solvers automatically use parallel computing whenever you call them with options.

To stop computing optimizations in parallel, set UseParallel to 'never', or set the Optimization Tool not to compute in parallel. To halt all parallel computation, enter

```
matlabpool close
```

Parallel Search Functions or Hybrid Functions

To have a patternsearch search function run in parallel, or a hybrid function for ga or simulannealbnd run in parallel, do the following.

1 Set up parallel processing as described in "Multicore Processors" on page 8-11 or "Processor Network" on page 8-12.

2 Ensure that your search function or hybrid function has the conditions outlined in these sections:

- "patternsearch Search Function" on page 8-14
- "Parallel Hybrid Functions" on page 8-15

patternsearch Search Function

patternsearch uses a parallel search function under the following conditions:

- CompleteSearch is 'on'.
- The search method is not @searchneldermead or custom.
- If the search method is a patternsearch poll method or Latin hypercube search, UseParallel is 'always'. Set at the command line with psoptimset:

```
options = psoptimset('UseParallel','always',...
    'CompleteSearch','on','SearchMethod',@GPSPositiveBasis2N);
```

Or you can use the Optimization Tool.
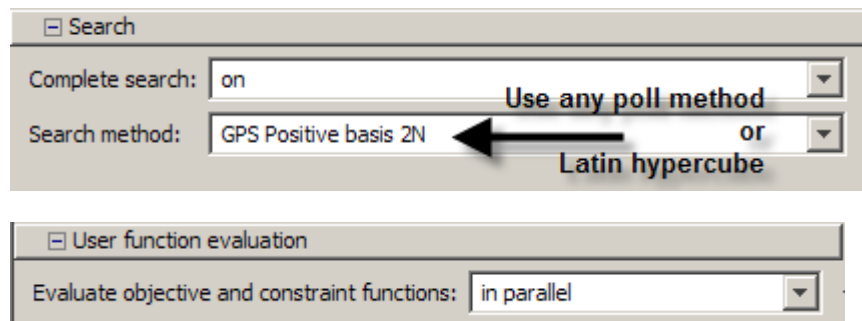
- If the search method is `ga`, the search method option structure has `UseParallel` set to `'always'`. Set at the command line with `psoptimset` and `gaoptimset`:

```
iterlim = 1; % iteration limit, specifies # ga runs
gaopt = gaoptimset('UseParallel','always');
options = psoptimset('SearchMethod',...
    {@searchga,iterlim,gaopt});
```

In the Optimization Tool, first create the `gaopt` structure as above, and then use these settings:
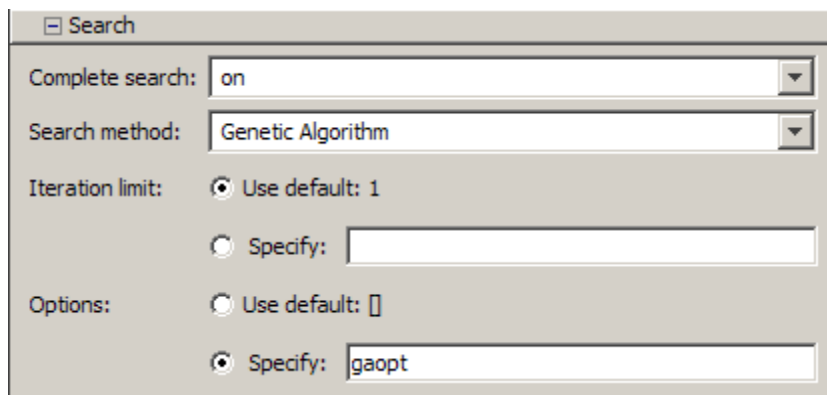
For more information about search functions, see "Using a Search Method" on page 4-55.

Parallel Hybrid Functions

`ga` and `simulannealbnd` can have other solvers run after or interspersed with their iterations. These other solvers are called hybrid functions. For information on using a hybrid function with `gamultiobj`, see "Parallel Computing with gamultiobj" on page 8-9. Both `patternsearch` and `fmincon` can be hybrid functions. You can set options so that `patternsearch` runs in parallel, or `fmincon` estimates gradients in parallel.

Set the options for the hybrid function as described in "Hybrid Function Options" on page 9-47 for `ga`, or "Hybrid Function Options" on page 9-57 for `simulannealbnd`. To summarize:

- If your hybrid function is `patternsearch`

  1 Create a `patternsearch` options structure:

  ```
  hybridopts = psoptimset('UseParallel','always',...
      'CompletePoll','on');
  ```

  2 Set the `ga` or `simulannealbnd` options to use `patternsearch` as a hybrid function:

  ```
  options = gaoptimset('UseParallel','always'); % for ga
  options = gaoptimset(options,...
      'HybridFcn',{@patternsearch,hybridopts});
  % or, for simulannealbnd:
  options = saoptimset('HybridFcn',{@patternsearch,hybridopts});
  ```

  Or use the Optimization Tool.
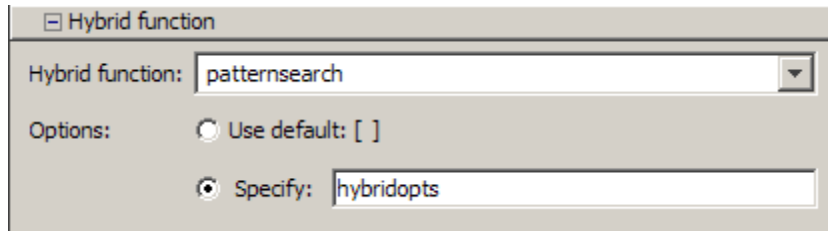
  For more information on parallel `patternsearch`, see "Pattern Search" on page 8-7.

- If your hybrid function is `fmincon`:

  1 Create a `fmincon` options structure:

  ```
  hybridopts = optimset('UseParallel','always',...
      'Algorithm','interior-point');
  % You can use any Algorithm except trust-region-reflective
  ```

  2 Set the `ga` or `simulannealbnd` options to use `fmincon` as a hybrid function:

  ```
  options = gaoptimset('UseParallel','always');
  options = gaoptimset(options,'HybridFcn',{@fmincon,hybridopts});
  ```

```
% or, for simulannealbnd:
options = saoptimset('HybridFcn',{@fmincon,hybridopts});
```

Or use the Optimization Tool.
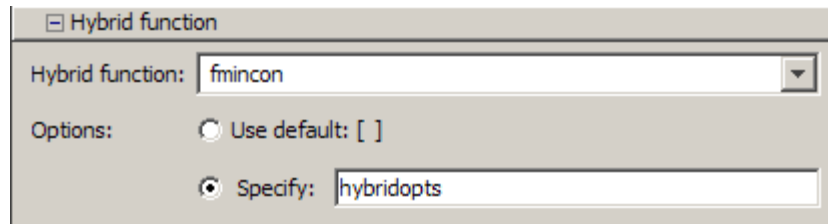
For more information on parallel `fmincon`, see "Parallel Computing for Optimization" in the Optimization Toolbox documentation.

9

Options Reference

GlobalSearch and MultiStart Properties (Options)

In this section...
"How to Set Properties" on page 9-2
"Properties of Both Objects" on page 9-2
"GlobalSearch Properties" on page 9-4
"MultiStart Properties" on page 9-6

How to Set Properties

To set a property of an existing `GlobalSearch` or `MultiStart` object, use dot addressing. For example, if `ms` is a `MultiStart` object, and you want to set the `Display` property to `'iter'`, enter

```
ms.Display = 'iter';
```

To set multiple properties simultaneously, use the constructor (`GlobalSearch` or `MultiStart`) with parameter-value pairs. For example, to set the `Display` property to `'iter'` and the `MaxTime` property to `100`, enter

```
ms = MultiStart(ms,'Display','iter','MaxTime',100);
```

For more information on setting properties, see "Changing Global Options" on page 3-57.

Properties of Both Objects

You can create a `MultiStart` object from a `GlobalSearch` object and vice-versa.

The syntax for creating a new object from an existing object is:

```
ms =  MultiStart(gs);
or
gs = GlobalSearch(ms);
```

The new object contains the properties that apply of the old object. This section describes those shared properties:

- "Display" on page 9-3
- "MaxTime" on page 9-3
- "StartPointsToRun" on page 9-3
- "TolX" on page 9-3
- "TolFun" on page 9-4

Display

Values for the Display property are:

- 'final' (default) — Summary results to command line after last solver run.
- 'off' — No output to command line.
- 'iter' — Summary results to command line after each local solver run.

MaxTime

The MaxTime property describes a tolerance on the number of seconds since the solver began its run. Solvers halt when they see MaxTime seconds have passed since the beginning of the run. Time means *wall clock* as opposed to processor cycles. The default is Inf.

StartPointsToRun

The StartPointsToRun property directs the solver to exclude certain start points from being run:

- all — Accept all start points.
- bounds — Reject start points that do not satisfy bounds.
- bounds-ineqs — Reject start points that do not satisfy bounds or inequality constraints.

TolX

The TolX property describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Set TolX to 0 to obtain the results of every local solver run. Set TolX to a larger value to

have fewer results. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance.

Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct.

TolFun

The `TolFun` property describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct.

GlobalSearch Properties

- "NumTrialPoints" on page 9-4
- "NumStageOnePoints" on page 9-5
- "MaxWaitCycle" on page 9-5
- "BasinRadiusFactor" on page 9-5
- "DistanceThresholdFactor" on page 9-5
- "PenaltyThresholdFactor" on page 9-5

NumTrialPoints

Number of potential start points to examine in addition to `x0` from the `problem` structure. `GlobalSearch` runs only those potential start points that pass several tests. For more information, see "GlobalSearch Algorithm" on page 3-37.

Default: `1000`

NumStageOnePoints

Number of start points in Stage 1. For details, see "Obtain Stage 1 Start Point, Run" on page 3-38.

Default: `200`

MaxWaitCycle

A positive integer tolerance appearing in several points in the algorithm.

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see "PenaltyThresholdFactor" on page 9-5).

- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see "BasinRadiusFactor" on page 9-5).

Default: `20`

BasinRadiusFactor

A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of $1 - \texttt{BasinRadiusFactor}$.

Default: `0.2`

DistanceThresholdFactor

A multiplier for determining whether a trial point is in an existing basin of attraction. For details, see "Examine Stage 2 Trial Point to See if fmincon Runs" on page 3-39. Default: `0.75`

PenaltyThresholdFactor

Determines increase in penalty threshold. For details, see React to Large Counter Values.

Default: `0.2`

MultiStart Properties

UseParallel

The `UseParallel` property determines whether the solver distributes start points to multiple processors:

- `'never'` (default) — Do not run in parallel.
- `'always'` — Run in parallel.

For the solver to run in parallel you must set up a parallel environment with `matlabpool`. For details, see "How to Use Parallel Processing" on page 8-11.

Pattern Search Options

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 4-38 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 9-9 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 documentation 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 classes of direct search algorithms: the generalized pattern search (GPS) algorithm, the generating set search (GSS) 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 (GPSPositiveBasis2N), 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 `GSS Positive basis 2N` pattern (`GSSPositiveBasis2N`) is similar to `GPS Positive basis 2N`, but adjusts the basis vectors to account for linear constraints. `GSS Positive basis 2N` is more efficient than `GPS Positive basis 2N` when the current point is near a linear constraint boundary.

- The `MADS Positive basis 2N` pattern (`MADSPositiveBasis2N`) 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 GSS Positive basis Np1 pattern (GSSPositiveBasisNp1) is similar to GPS Positive basis Np1, but adjusts the basis vectors to account for linear constraints. GSS Positive basis Np1 is more efficient than GPS Positive basis Np1 when the current point is near a linear constraint boundary.

- The MADS Positive basis Np1 pattern (MADSPositiveBasisNp1) 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 4-42.

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`, `GSS Positive basis Np1`, `GSS 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 2N` (`@GPSPositiveBasis2N`)
- `GPS Positive basis Np1` (`@GPSPositiveBasisNp1`)
- `GSS Positive basis 2N` (`@GSSPositiveBasis2N`)
- `GSS Positive basis Np1` (`@GSSPositiveBasisNp1`)
- `MADS Positive basis 2N` (`@MADSPositiveBasis2N`)
- `MADS Positive basis Np1` (`@MADSPositiveBasisNp1`)

- 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.
  patternsearch generates each point for the search as follows. For each
  component, take a random permutation of the vector [1,2,...,k] minus
  rand(1,k), divided by k. (k is the number of points.) This yields k points,
  with each component close to evenly spaced. The resulting points are then
  scaled to fit any bounds. Latin hypercube uses default bounds of -1 and 1.

  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** — The **Design level** is the number of points
  patternsearch searches, a positive integer. The default for **Design
  level** is 15 times the number of dimensions.

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 4-55 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 a GPS or GSS algorithm is selected as the Poll or Search method. The default value is `Inf`. MADS uses a maximum size of `1`.

Accelerator (`MeshAccelerator`) specifies whether, when the mesh size is small, the **Contraction factor** is multiplied by `0.5` after each unsuccessful iteration. **Accelerator** can have the values `On` or `Off`, the default. **Accelerator** applies to the GPS and GSS algorithms.

Rotate (`MeshRotate`) is only applied when **Poll method** is set to `GPS Positive basis Np1` or `GSS 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`, `GSS 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 or GSS method is selected as the Poll or Search method. MADS uses a factor of `4.0`.

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 or GSS method is selected as the Poll or Search method. MADS uses a factor of `0.25`.

See "Mesh Expansion and Contraction" on page 4-59 for more information.

Algorithm Settings

Algorithm settings refer to constraint parameters. For details, see
"Description of the Nonlinear Constraint Solver" on page 4-24.

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.

Bind tolerance (`TolBind`) specifies the tolerance for the distance from the current point to the boundary of the feasible region with respect to linear constraints. This means `Bind tolerance` specifies when a linear constraint is active. `Bind tolerance` is not a stopping criterion. Active linear constraints change the pattern of points `patternsearch` uses for polling or searching. `patternsearch` always uses points that satisfy linear constraints to within `Bind tolerance`. The default value of `Bind tolerance` is `1e-3`.

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 4-65 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 GPS and GSS algorithms stop if the mesh size becomes smaller than **Mesh tolerance**. MADS 2N stops when the mesh size becomes smaller than `TolMesh^2`. MADS Np1 stops when the mesh size becomes smaller than `(TolMesh/nVar)^2`, where `nVar` is the number of elements of `x0`. The default value of `TolMesh` 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.

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 `TolFun`, the algorithm halts. The default value is `1e-6`.

See "Setting Tolerances for the Solver" on page 4-68 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.

    - To pass extra parameters in the output function, use "Anonymous Functions".

    - For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

    At the command line, set

        options = psoptimset('OutputFcn',@myfun);

    For multiple output functions, enter a cell array:

```
options = psoptimset('OutputFcn',{@myfun1,@myfun2,...});
```

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 documentation 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 and Parallel Options (User Function Evaluation)

You can choose to have your objective and constraint functions evaluated in serial, parallel, or in a vectorized fashion. These options are available in the **User function evaluation** section of the **Options** pane of the Optimization Tool, or by setting the `'Vectorized'` and `'UseParallel'` options with `psoptimset`.

- When **Evaluate objective and constraint functions** (`'Vectorized'`) is **in serial** (`'Off'`), `patternsearch` calls the objective function on one point at a time as it loops through all of the mesh points. (At the command line, this assumes `'UseParallel'` is at its default value of `'never'`.)

- When **Evaluate objective and constraint functions** (`'Vectorized'`) is **vectorized** (`'On'`), `patternsearch` 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.

- When **Evaluate objective and constraint functions** (UseParallel) is **in parallel** (`'always'`) patternsearch calls the objective function in parallel, using the parallel environment you established (see "How to Use Parallel Processing" on page 8-11). At the command line, set UseParallel to `'never'` to compute serially.

Note You cannot simultaneously use vectorized and parallel computations. If you set `'UseParallel'` to `'always'` and `'Vectorized'` to `'On'`, patternsearch evaluates your objective and constraint functions in a vectorized manner, not in parallel.

How Objective and Constraint Functions Are Evaluated

	`Vectorized = Off`	`Vectorized = On`
UseParallel = `'Never'`	Serial	Vectorized
UseParallel = `'Always'`	Parallel	Vectorized

Options Table for Pattern Search Algorithms

Option Availability Table for All 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	All

Option Availability Table for All Algorithms (Continued)

Option	Description	Algorithm Availability
	stochastic, it is advised not to use this option.	
CacheSize	Size of the cache, in number of points.	All
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'.	All
CompletePoll	Complete poll around current iterate. Evaluate all the points in a poll step.	All
CompleteSearch	Complete search around current iterate. Evaluate all the points in a search step.	All
Display	Level of display to Command Window.	All
InitialMeshSize	Initial mesh size used in pattern search algorithms.	All
InitialPenalty	Initial value of the penalty parameter.	All
MaxFunEvals	Maximum number of objective function evaluations.	All
MaxIter	Maximum number of iterations.	All

Option Availability Table for All Algorithms (Continued)

Option	Description	Algorithm Availability
MaxMeshSize	Maximum mesh size used in a poll/search step.	GPS and GSS
MeshAccelerator	Accelerate mesh size contraction.	GPS and GSS
MeshContraction	Mesh contraction factor, used when iteration is unsuccessful.	GPS and GSS
MeshExpansion	Mesh expansion factor, expands mesh when iteration is successful.	GPS and GSS
MeshRotate	Rotate the pattern before declaring a point to be optimum.	GPS Np1 and GSS Np1
OutputFcn	User-specified function that a pattern search calls at each iteration.	All
PenaltyFactor	Penalty update parameter.	All
PlotFcn	Specifies function to plot at runtime.	All
PlotInterval	Specifies that plot functions will be called at every interval.	All
PollingOrder	Order in which search directions are polled.	GPS and GSS
PollMethod	Polling strategy used in pattern search.	All
ScaleMesh	Automatic scaling of variables.	All

Option Availability Table for All Algorithms (Continued)

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

Genetic Algorithm Options

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 5-40 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** (`@gaplotstopping`) 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 9-32.

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 9-32 describes the fields of `state`.

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

"Passing Extra Parameters" in the Optimization Toolbox documentation 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 type (`PopulationType`) specifies the type of the input to the fitness function. Types and their restrictions:

- 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. For **Creation function** (`CreationFcn`) and **Mutation function** (`MutationFcn`), use Uniform (`mutationuniform`) or Custom. For **Crossover function** (`CrossoverFcn`), use Scattered (`@crossoverscattered`), Single point (`@crossoversinglepoint`), Two point (`@crossovertwopoint`), or Custom. You cannot use a **Hybrid function** or **Nonlinear constraint function**.

- Custom — For **Crossover function** and **Mutation function**, use Custom. For **Creation function**, either use Custom, or provide an **Initial population**. You cannot use a **Hybrid function** or **Nonlinear constraint function**.

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 documentation 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 5-50 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 documentation explains how to provide additional parameters to the function.

See "Fitness Scaling" on page 5-59 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 documentation explains how to provide additional parameters to the function.

See "Selection" on page 5-62 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 5-66 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 9-32 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 documentation 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 crossover 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 documentation 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 refer to the nonlinear constraint solver. For details, see "Description of the Nonlinear Constraint Solver" on page 5-27.

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 5-76 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 5-80 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.
- To pass extra parameters in the output function, use "Anonymous Functions".
- For multiple output functions, enter a cell array of output function handles: {@myfun1,@myfun2,...}.

At the command line, set

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

For multiple output functions, enter a cell array:

```
options = gaoptimset('OutputFcn',{@myfun1,@myfun2,...});
```

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 9-32 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 documentation 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 and Parallel Options (User Function Evaluation)

You can choose to have your fitness and constraint functions evaluated in serial, parallel, or in a vectorized fashion. These options are available in the **User function evaluation** section of the **Options** pane of the Optimization Tool, or by setting the 'Vectorized' and 'UseParallel' options with gaoptimset.

- When **Evaluate fitness and constraint functions** ('Vectorized') is **in serial** ('Off'), ga calls the fitness function on one individual at a time as it loops through the population. (At the command line, this assumes 'UseParallel' is at its default value of 'never'.)

- When **Evaluate fitness and constraint functions** ('Vectorized') is **vectorized** ('On'), ga calls the fitness function on the entire population at once, i.e., in a single call to the fitness function.

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

  See "Vectorizing the Fitness Function" on page 5-81 for an example.

- When **Evaluate fitness and constraint functions** (UseParallel) is **in parallel** ('always'), ga calls the fitness function in parallel, using the

parallel environment you established (see "How to Use Parallel Processing" on page 8-11). At the command line, set `UseParallel` to `'never'` to compute serially.

Note You cannot simultaneously use vectorized and parallel computations. If you set `'UseParallel'` to `'always'` and `'Vectorized'` to `'On'`, ga evaluates your fitness and constraint functions in a vectorized manner, not in parallel.

How Fitness and Constraint Functions Are Evaluated

	Vectorized = Off	**Vectorized = On**
UseParallel = 'Never'	Serial	Vectorized
UseParallel = 'Always'	Parallel	Vectorized

Simulated Annealing Options

saoptimset At The Command Line

Specify options by creating an `options` structure using the `saoptimset` function as follows:

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

See "Setting Options for simulannealbnd at the Command Line" on page 6-15 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 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` 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 9-9 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

- 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. The syntax is:

      temperature = myfun(optimValues,options)

  where optimValues is a structure described in "Structure of the Plot Functions" on page 9-54. options is either the structure created with saoptimset, or the structure of default options, if you did not create an options structure. For example, the function temperaturefast is:

      temperature = options.InitialTemperature./optimValues.k;

- 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 simulannealbnd are:

- DataType — Type of data to use in the objective function. Choices:
  - 'double' (default) — A vector of type double.
  - 'custom' — Any other data type. You must provide a 'custom' annealing function. You cannot use a hybrid function.

- 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 9-59, and `problem` is a structure containing the following information:

- `objective`: function handle to the objective function
- `x0`: the start point
- `nvar`: number of decision variables
- `lb`: lower bound on decision variables
- `ub`: upper bound on decision variables

For example, the current position is `optimValues.x`, and the current objective function value is `problem.objective(optimValues.x)`.

`AcceptanceFcn` — Function used to determine whether a new point is accepted or not. The choices are:

- `@acceptancesa` — Simulated annealing acceptance function. The default for `simulannealbnd`.

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

      acceptpoint = myfun(optimValues,newx,newfval);

  where `optimValues` is a structure described in "Structure of the Output Function" on page 9-59, `newx` is the point being evaluated for acceptance, and `newfval` is the objective function at `newx`. `acceptpoint` is a Boolean, with value `true` to accept `newx`, and `false` to reject `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 simulannealbnd options structure as follows:

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

hybridopts must exist before you set options.

See "Using a Hybrid Function" on page 5-76 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 9-59.

Using the Optimization Tool:

- Specify **Output function** as `@myfun`, where `myfun` is the name of your function.

- To pass extra parameters in the output function, use "Anonymous Functions".

- For multiple output functions, enter a cell array of output function handles: `{@myfun1,@myfun2,...}`.

At the command line:

-

    options = saoptimset('OutputFcns',@myfun);

- For multiple output functions, enter a cell array:

    options = saoptimset('OutputFcn',{@myfun1,@myfun2,...});

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
- 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

"Passing Extra Parameters" in the Optimization Toolbox documentation explains how to provide additional parameters to the output function.

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

10

Function Reference

GlobalSearch

createOptimProblem	Create optimization problem structure
run (GlobalSearch)	Find global minimum

MultiStart

createOptimProblem	Create optimization problem structure
list (CustomStartPointSet)	List custom start points in set
list (RandomStartPointSet)	Generate start points
run (MultiStart)	Run local solver from multiple points

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

saoptimget	Values of simulated annealing options structure
saoptimset	Create simulated annealing options structure
simulannealbnd	Find unconstrained or bound-constrained minimum of function of several variables using simulated annealing algorithm

11

Class Reference

Functions — Alphabetical List

createOptimProblem

Purpose	Create optimization problem structure
Syntax	*problem* = createOptimProblem('*solverName*') *problem* = createOptimProblem('*solverName*','*ParameterName*', *ParameterValue*,...)
Description	*problem* = createOptimProblem('*solverName*') creates an empty optimization problem structure for the *solverName* solver. *problem* = createOptimProblem('*solverName*','*ParameterName*',*ParameterValue*,...) accepts one or more comma-separated parameter name/value pairs. Specify *ParameterName* inside single quotes.
Input Arguments	solverName Name of the solver. For a GlobalSearch problem, use 'fmincon'. For a MultiStart problem, use 'fmincon', 'fminunc', 'lsqcurvefit' or 'lsqnonlin'.

Parameter Name/Value Pairs

Aeq

    Matrix for linear equality constraints. The constraints have the form:

    Aeq x = beq

Aineq

    Matrix for linear inequality constraints. The constraints have the form:

    Aineq x ≤ bineq

beq

    Vector for linear equality constraints. The constraints have the form:

    Aeq x = beq

`bineq`

Vector for linear inequality constraints. The constraints have the form:

`Aineq x ≤ bineq`

`lb`

Vector of lower bounds.

`nonlcon`

Function handle to the nonlinear constraint function. The constraint function must accept a vector `x` and return two vectors: `c`, the nonlinear inequality constraints, and `ceq`, the nonlinear equality constraints. If one of these constraint functions is empty, `nonlcon` must return `[]` for that function.

If the `GradConstr` option is `'on'`, then in addition `nonlcon` must return two additional outputs, `gradc` and `gradceq`. The `gradc` parameter is a matrix with one column for the gradient of each constraint, as is `gradceq`.

For more information, see "Constraints" on page 2-6.

`objective`

Function handle to the objective function. For all solvers except `lsqnonlin` and `lsqcurvefit`, the objective function must accept a vector `x` and return a scalar. If the `GradObj` option is `'on'`, then the objective function must return a second output, a vector, representing the gradient of the objective. For `lsqnonlin`, the objective function must accept a vector `x` and return a vector. `lsqnonlin` sums the squares of the objective function values. For `lsqcurvefit`, the objective function must accept two inputs, `x` and `xdata`, and return a vector.

For more information, see "Computing Objective Functions" on page 2-2.

`options`

createOptimProblem

Options structure. Create this structure with `optimset`, or by exporting from the Optimization Tool.

ub

Vector of upper bounds.

x0

A vector, a potential starting point for the optimization. Gives the dimensionality of the problem.

xdata

Vector of data points for `lsqcurvefit`.

ydata

Vector of data points for `lsqcurvefit`.

Output Arguments

problem

Optimization problem structure.

Examples

Create a problem structure using Rosenbrock's function as objective (see "Using a Hybrid Function" on page 5-76), the `interior-point` algorithm for `fmincon`, and bounds with absolute value 2:

```
anonrosen = @(x)(100*(x(2) - x(1)^2)^2 + (1-x(1))^2);
opts = optimset('Algorithm','interior-point');
problem = createOptimProblem('fmincon','x0',randn(2,1),...
    'objective',anonrosen,'lb',[-2;-2],'ub',[2;2],...
    'options',opts);
```

Alternatives

You can create a problem structure by exporting from the Optimization Tool (`optimtool`), as described in "Exporting from the Optimization Tool" on page 3-8.

See Also

optimtool | MultiStart | GlobalSearch

Tutorials

• "Create a Problem Structure" on page 3-4

Purpose   User-supplied start points

Description   An object wrapper of a matrix whose rows represent start points for
                   MultiStart.

Construction   *tpoints* = CustomStartPointSet(*ptmatrix*) generates a
                   CustomStartPointSet object from the *ptmatrix* matrix. Each row of
                   ptmatrix represents one start point.

Properties   DimStartPoints

                   Dimension of each start point, a read-only property.
                   DimStartPoints is the number of columns in *ptmatrix*.

                   DimStartPoints should be the same as the number of elements in
                   problem.x0, the problem structure you pass to run.

                   NumStartPoints

                   Number of start points, a read-only property. This is the number
                   of rows in *ptmatrix*.

Methods

                   list   List custom start points in set

Copy   Value. To learn how value classes affect copy operations, see Copying
Semantics   Objects in the MATLAB Programming Fundamentals documentation.

Examples   Create a CustomStartPointSet object with 40 three-dimensional rows.
                   Each row represents a normally distributed random variable with mean
                   [10,10,10] and variance diag([4,4,4]):

                   ```
                   fortypts = 10*ones(40,3) + 4*randn(40,3); % a matrix
                   startpts = CustomStartPointSet(fortypts);
                   ```

                   startpts is the fortypts matrix in an object wrapper.

CustomStartPointSet class

Get the `fortypts` matrix from the `startpts` object of the previous example:

```
fortypts = list(startpts);
```

See Also   RandomStartPointSet | MultiStart | list

Tutorials   • "CustomStartPointSet Object for Start Points" on page 3-17

How To   • Class Attributes

• Property Attributes

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 documentation.

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 nvars variables, then

- A is a matrix of size m-by-nvars.

- 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 nvars variables, then

- Aeq is a matrix of size r-by-nvars.

- 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
rngstate	Optional field to reset the state of the random number generator
solver	'ga'
options	Options structure created using gaoptimset or the Optimization Tool

Create the structure problem by exporting a problem from Optimization Tool, as described in "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

[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:

- rngstate — The state of the MATLAB random number generator,
  just before the algorithm started. You can use the values in rngstate
  to reproduce the output of ga. See "Reproducing Your Results" on
  page 5-44.

- 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

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$

$$x_1 \ge 0, \ x_2 \ge 0,$$

the following code finds the minimum of the function, lincontest6, that is provided 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`

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 $A*x \le b$. 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 |

| rngstate | Optional field to reset the state of the random number generator |
|----------|------------------------------------------------------------------|
| 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 documentation.

[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 |
|---|---|
| rngstate | State of the MATLAB random number generator, just before the algorithm started. You can use the values in rngstate to reproduce the output of ga. See "Reproducing Your Results" on page 5-44. |
| generations | Total number of generations, excluding HybridFcn iterations |
| funccount | Total number of function evaluations |
| maxconstraint | Maximum constraint violation, if any |
| 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**     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 a
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.

Algorithm   `gamultiobj` uses a controlled elitist genetic algorithm (a variant of NSGA-II [1]). An elitist GA always favors individuals with better fitness value (rank). A controlled elitist GA also favors individuals that can help increase the diversity of the population even if they have a lower fitness value. It is important to maintain the diversity of population for convergence to an optimal Pareto front. Diversity is maintained by controlling the elite members of the population as the algorithm progresses. Two options, `ParetoFraction` and `DistanceFcn`, control the elitism. `ParetoFraction` limits the number of individuals on the Pareto front (elite members). The distance function, selected by `DistanceFcn`, helps to maintain diversity on a front by favoring individuals that are relatively far away on the front.

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
                 9-29.

                 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 called 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 9-29 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 | @gacreationlinearfeasible
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. Not used in gamultiobj.	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}

Option	Description	Values
InitialPopulation	Initial population used to seed the genetic algorithm; can be partial	Matrix | {[]}
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 9-29.

ga, gamultiobj, gaoptimget

GlobalOptimSolution class

Purpose   Optimization solution

Description   Information on a local minimum, including location, objective function value, and start point or points that lead to the minimum.

GlobalSearch and MultiStart generate a vector of GlobalOptimSolution objects. The vector is ordered by objective function value, from lowest (best) to highest (worst).

Construction   When you run them, GlobalSearch and MultiStart create GlobalOptimSolution objects as output.

Properties   Exitflag

An integer describing the result of the local solver run.

For the meaning of the exit flag, see the description in the appropriate local solver function reference page:

- fmincon
- fminunc
- lsqcurvefit
- lsqnonlin

Fval

Objective function value at the solution.

Output

Output structure returned by the local solver.

X

Solution point, with the same dimensions as the initial point.

X0

Cell array of start points that led to the solution.

Copy Semantics	Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.
Examples	Use `MultiStart` to create a vector of `GlobalOptimSolution` objects:

```
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,30);
```

allmins is the vector of `GlobalOptimSolution` objects:

```
allmins

allmins =

  1x30 GlobalOptimSolution

  Properties:
    X
    Fval
    Exitflag
    Output
    X0
```

See Also	MultiStart	MultiStart.run	GlobalSearch	GlobalSearch.run
Tutorials	• "Multiple Solutions" on page 3-24
	• "Example: Visualizing the Basins of Attraction" on page 3-32
How To	• Class Attributes
	• Property Attributes

GlobalSearch class

Purpose   Find global minimum

Description   A `GlobalSearch` object contains properties (options) that affect how the `run` method searches for a global minimum, or generates a `GlobalOptimSolution` object.

Construction   *gs* = GlobalSearch constructs a new global search optimization solver with its properties set to the defaults.

gs = GlobalSearch(*'PropertyName'*,*PropertyValue*,...) constructs the object using options, specified as property name and value pairs.

gs = GlobalSearch(*oldgs*,*'PropertyName'*,*PropertyValue*,...) constructs a copy of the GlobalSearch solver *oldgs*. The *gs* object has the named properties altered with the specified values.

gs = GlobalSearch(*ms*) constructs *gs*, a GlobalSearch solver, with common parameter values from the *ms* MultiStart solver.

Properties   BasinRadiusFactor

        A basin radius decreases after `MaxWaitCycle` consecutive start points are within the basin. The basin radius decreases by a factor of 1 – `BasinRadiusFactor`.

        Set `BasinRadiusFactor` to `0` to disable updates of the basin radius.

        Default: `0.2`

    Display

        Detail level of iterative display. Possible values:

- `'final'` — Report summary results after `run` finishes.

- `'iter'` — Report results after the initial `fmincon` run, after Stage 1, after every 200 start points, and after every run of `fmincon`, in addition to the final summary.

- `'off'` — No display.

  Default: `'final'`

`DistanceThresholdFactor`

A multiplier for determining whether a trial point is in an existing basin of attraction. For details, see "Examine Stage 2 Trial Point to See if fmincon Runs" on page 3-39.

Default: `0.75`

`MaxTime`

Time in seconds for a run. `GlobalSearch` halts when it sees `MaxTime` seconds have passed since the beginning of the run.

Default: `Inf`

`MaxWaitCycle`

A positive integer tolerance appearing in several points in the algorithm:

- If the observed penalty function of `MaxWaitCycle` consecutive trial points is at least the penalty threshold, then raise the penalty threshold (see `PenaltyThresholdFactor`).

- If `MaxWaitCycle` consecutive trial points are in a basin, then update that basin's radius (see `BasinRadiusFactor`).

Default: `20`

`NumStageOnePoints`

Number of start points in Stage 1. For details, see "Obtain Stage 1 Start Point, Run" on page 3-38.

Default: `200`

GlobalSearch class

NumTrialPoints

Number of potential start points to examine in addition to x0 from the problem structure. GlobalSearch runs only those potential start points that pass several tests. For more information, see "GlobalSearch Algorithm" on page 3-37.

Default: 1000

PenaltyThresholdFactor

Determines increase in the penalty threshold. For details, see React to Large Counter Values.

Default: 0.2

StartPointsToRun

Directs the solver to exclude certain start points from being run:

- all — Accept all start points.

- bounds — Reject start points that do not satisfy bounds.

- bounds-ineqs — Reject start points that do not satisfy bounds or inequality constraints.

GlobalSearch checks the StartPointsToRun property only during Stage 2 of the GlobalSearch algorithm (the main loop). For more information, see "GlobalSearch Algorithm" on page 3-37.

Default: 'all'

TolFun

Describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Solvers consider two solutions identical if they are within TolX distance of each other and have objective function values within TolFun of each other. If both conditions are not

met, solvers report the solutions as distinct. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Default: `1e-6`

TolX

Describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolX` to `0` to obtain the results of every local solver run. Set `TolX` to a larger value to have fewer results.

Default: `1e-6`

Methods

run	Find global minimum

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Solve a problem using a default `GlobalSearch` object:

```
opts = optimset('Algorithm','interior-point');
problem = createOptimProblem('fmincon','objective',...
 @(x) x.^2 + 4*sin(5*x),'x0',3,'lb',-5,'ub',5,'options',opts);
gs = GlobalSearch;
[x,f] = run(gs,problem)
```

Algorithm

A detailed description of the algorithm appears in "GlobalSearch Algorithm" on page 3-37. Ugray et al. [1] describes both the algorithm and the scatter-search method of generating trial points.

GlobalSearch class

References   [1] Ugray, Zsolt, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization.* INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

See Also   MultiStart | createOptimProblem | GlobalOptimSolution

Tutorials   • Chapter 3, "Using GlobalSearch and MultiStart"

                               • "GlobalSearch and MultiStart Examples" on page 3-63

How To   • Class Attributes

                               • Property Attributes

Purpose List custom start points in set

Syntax *tpts* = list(*CS*)

Description *tpts* = list(*CS*) returns the matrix of start points in the *CS*
CustomStartPoints object.

Input Arguments CS

A CustomStartPointSet object.

Output Arguments tpts

Matrix of start points. The rows of tpts represent the start points.

Examples Create a CustomStartPointSet containing 40 seven-dimensional
normally distributed points, then use list to get the matrix of points
from the object:

```
startpts = randn(40,7) % 40 seven-dimensional start points
cs = CustomStartPointSet(startpts); % cs is an object
startpts2 = list(cs) % startpts2 = startpts
```

See Also CustomStartPointSet | createOptimProblem

Tutorials • "CustomStartPointSet Object for Start Points" on page 3-17

RandomStartPointSet.list

Purpose	Generate start points
Syntax	*points* = list(*RandSet*,*problem*)
Description	*points* = list(*RandSet*,*problem*) generates pseudorandom start points using the parameters in the *RandSet* RandomStartPointSet object, and information from the *problem* problem structure.

Input Arguments

RandSet

> A RandomStartPointSet object. This contains parameters for generating the points: number of points, random stream, and artificial bounds.

problem

> An optimization problem structure. list generates points uniformly within the bounds of the problem structure. If a component is unbounded, list uses the artificial bounds from RandSet. list takes the dimension of the points from the x0 field in problem.

Output Arguments

points

> A k-by-n matrix. The number of rows k is the number of start points that RandSet specifies. The number of columns n is the dimension of the start points. n is equal to the number of elements in the x0 field in problem. The MultiStart algorithm uses each row of points as an initial point in an optimization.

Examples

Create a matrix representing 40 seven-dimensional start points:

```
rs = RandomStartPointSet('NumStartPoints',40); % 40 points
problem = createOptimProblem('fminunc','x0',ones(7,1),...
    'objective',@rosenbrock);
ptmatrix = list(rs,problem); % matrix values between
    % -1000 and 1000 since those are the default bounds
    % for unconstrained dimensions
```

Algorithm   The `list` method generates a pseudorandom random matrix. Each row of the matrix represents a start point to run. `list` generates points that satisfy the bounds in `problem`. If `lb` is the vector of lower bounds, `ub` is the vector of upper bounds, there are `n` dimensions in a point, and there are `k` rows in the matrix, the random matrix is

```
lb + (ub - lb).*rand(k,n)
```

- If a component has no bounds, RandomStartPointSet uses a lower bound of `-ArtificialBound`, and an upper bound of `ArtificialBound`.

- If a component has a lower bound `lb`, but no upper bound, RandomStartPointSet uses an upper bound of `lb + 2*ArtificialBound`.

- Similarly, if a component has an upper bound `ub`, but no lower bound, RandomStartPointSet uses a lower bound of `ub - 2*ArtificialBound`.

The default value of `ArtificialBound` is `1000`.

See Also   RandomStartPointSet | createOptimProblem | MultiStart

Tutorials   • "RandomStartPointSet Object for Start Points" on page 3-17

MultiStart class

Purpose　　　Find multiple local minima

Description　A `MultiStart` object contains properties (options) that affect how the `run` method repeatedly runs a local solver, or generates a `GlobalOptimSolution` object.

Construction　*MS* = MultiStart constructs *MS*, a `MultiStart` solver with its properties set to the defaults.

MS = MultiStart('*PropertyName*',*PropertyValue*,...) constructs *MS* using options, specified as property name and value pairs.

MS = MultiStart(*oldMS*,'*PropertyName*',*PropertyValue*,...) creates a copy of the *oldMS* `MultiStart` solver, with the named properties changed to the specified values.

MS = MultiStart(*GS*) constructs *MS*, a `MultiStart` solver, with common parameter values from the *GS* `GlobalSearch` solver.

Properties　　Display

Detail level of the output to the Command Window:

- 'final' — Report summary results after `run` finishes.
- 'iter' — Report results after each local solver run, in addition to the final summary.
- 'off' — No display.

Default: final

MaxTime

Tolerance on the time `MultiStart` runs. `MultiStart` and its local solvers halt when they see MaxTime seconds have passed since the beginning of the run. Time means *wall clock* as opposed to processor cycles.

Default: Inf

StartPointsToRun

Directs the solver to exclude certain start points from being run:

- `all` — Accept all start points.
- `bounds` — Reject start points that do not satisfy bounds.
- `bounds-ineqs` — Reject start points that do not satisfy bounds or inequality constraints.

Default: `all`

TolFun

Describes how close two objective function values must be for solvers to consider them identical for creating the vector of local solutions. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolFun` to `0` to obtain the results of every local solver run. Set `TolFun` to a larger value to have fewer results.

Default: `1e-6`

TolX

Describes how close two points must be for solvers to consider them identical for creating the vector of local solutions. Solvers compute the distance between a pair of points with `norm`, the Euclidean distance. Solvers consider two solutions identical if they are within `TolX` distance of each other and have objective function values within `TolFun` of each other. If both conditions are not met, solvers report the solutions as distinct. Set `TolX` to `0` to obtain the results of every local solver run. Set `TolX` to a larger value to have fewer results.

Default: `1e-6`

MultiStart class

UseParallel

Distribute local solver calls to multiple processors:

- always — Distribute the local solver calls to multiple processors.

- never — Cannot not run in parallel.

Default: never

Methods

run	Run local solver from multiple points

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Run MultiStart on 20 instances of a problem using the fmincon sqp algorithm:

```
opts = optimset('Algorithm','sqp');
problem = createOptimProblem('fmincon','objective',...
 @(x) x.^2 + 4*sin(5*x),'x0',3,'lb',-5,'ub',5,'options',opts);
ms = MultiStart;
[x,f] = run(ms,problem,20)
```

Algorithm

A detailed description of the algorithm appears in "MultiStart Algorithm" on page 3-41.

See Also

GlobalSearch | createOptimProblem | RandomStartPointSet | CustomStartPointSet | GlobalOptimSolution

Tutorials

- Chapter 3, "Using GlobalSearch and MultiStart"

- "GlobalSearch and MultiStart Examples" on page 3-63

- Chapter 8, "Parallel Processing"

How To
- Class Attributes
- Property Attributes

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 documentation.

---

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 $LB \le x \le UB$. 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:

- `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`

- `rngstate` — Optional field to reset the state of the random number generator

Create the structure `problem` by exporting a problem from the Optimization Tool, as described in "Importing and Exporting Your Work" in the Optimization Toolbox documentation.

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

- `iterations` — 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

$$\begin{bmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix},$$

$$x_1 \geq 0, \quad x_2 \geq 0,$$

the following code finds the minimum of the function, lincontest6, that is provided 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] Audet, Charles and J. E. Dennis Jr. "Analysis of Generalized Pattern Searches." *SIAM Journal on Optimization*, Volume 13, Number 3, 2003, pp. 889–903.

[2] 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, 1997, pp. 261–288.

[3] Abramson, Mark A. *Pattern Search Filter Algorithms for Mixed Variable General Constrained Optimization Problems*. Ph.D. Thesis, Department of Computational and Applied Mathematics, Rice University, August 2002.

[4] Abramson, Mark A., Charles Audet, J. E. Dennis, Jr., and Sebastien Le Digabel. "ORTHOMADS: A deterministic MADS instance with orthogonal directions." *SIAM Journal on Optimization*, Volume 20, Number 2, 2009, pp. 948–966.

[5] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "Optimization by direct search: new perspectives on some classical and modern methods." *SIAM Review*, Volume 45, Issue 3, 2003, pp. 385–482.

[6] Kolda, Tamara G., Robert Michael Lewis, and Virginia Torczon. "A generating set direct search augmented Lagrangian algorithm for optimization with a combination of general and linear constraints." Technical Report SAND2006-5315, Sandia National Laboratories, August 2006.

[7] Lewis, Robert Michael, Anne Shepherd, and Virginia Torczon. "Implementing generating set search methods for linearly constrained minimization." *SIAM Journal on Scientific Computing*, Volume 29, Issue 6, 2007, pp. 2507–2530.

See Also   `optimtool`, `psoptimget`, `psoptimset`, `ga`, `simulannealbnd`

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 9-7. 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 9-7 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}

Option	Description	Values
PollingOrder	Order of poll directions in pattern search	`'Random'` | `'Success'` | `{'Consecutive'}`
PollMethod	Polling strategy used in pattern search	`{'GPSPositiveBasis2N'}` | `'GPSPositiveBasisNp1'` | `'GSSPositiveBasis2N'` | `'GSSPositiveBasisNp1'` | `'MADSPositiveBasis2N'` | `'MADSPositiveBasisNp1'`
ScaleMesh	Automatic scaling of variables	`{'on'}` | `'off'`
SearchMethod	Type of search used in pattern search	`@GPSPositiveBasis2N` | `@GPSPositiveBasisNp1` | `@GSSPositiveBasis2N` | `@GSSPositiveBasisNp1` | `@MADSPositiveBasis2N` | `@MADSPositiveBasisNp1` | `@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`

GlobalSearch.run

Purpose	Find global minimum
Syntax	*x* = run(*gs*,*problem*) [*x*,*fval*] = run(*gs*,*problem*) [*x*,*fval*,*exitflag*] = run(*gs*,*problem*) [*x*,*fval*,*exitflag*,*output*] = run(*gs*,*problem*) [*x*,*fval*,*exitflag*,*output*,*solutions*] = run(*gs*,*problem*)

Description

x = run(*gs*,*problem*) finds a point *x* that solves the optimization problem described in the *problem* structure.

[*x*,*fval*] = run(*gs*,*problem*) returns the value of the objective function in *problem* at the point *x*.

[*x*,*fval*,*exitflag*] = run(*gs*,*problem*) returns an exit flag describing the results of the multiple local searches.

[*x*,*fval*,*exitflag*,*output*] = run(*gs*,*problem*) returns an output structure describing the iterations of the run.

[*x*,*fval*,*exitflag*,*output*,*solutions*] = run(*gs*,*problem*) returns a vector of solutions containing the distinct local minima found during the run.

Input Arguments

gs

    A GlobalSearch object.

problem

    Problem structure. Create problem with createOptimProblem or by exporting a problem structure from the Optimization Tool. problem must contain at least the following fields:

- solver
- objective
- x0

- options — Both `createOptimProblem` and the Optimization Tool always include an `options` field in the `problem` structure.

Output Arguments

x

   Minimizing point of the objective function.

fval

   Objective function value at the minimizer x.

exitflag

   Describes the results of the multiple local searches. Values are:

2	At least one local minimum found. Some runs of the local solver converged.
1	At least one local minimum found. All runs of the local solver converged.
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the `MaxIter` or `MaxFunEvals` tolerances.
-1	One or more local solver runs stopped by the local solver output or plot function.
-2	No feasible local minimum found.
-5	`MaxTime` limit exceeded.
-8	No solution found. All runs had local solver exit flag `-2` or smaller, not all equal `-2`.
-10	Failures encountered in user-provided functions.

output

   A structure describing the iterations of the run. Fields in the structure:

funcCount	Number of function evaluations.
localSolverIncomplete	Number of local solver runs with 0 exit flag.
localSolverNoSolution	Number of local solver runs with negative exit flag.
localSolverSuccess	Number of local solver runs with positive exit flag.
localSolverTotal	Total number of local solver runs.
message	Exit message.

solutions

A vector of GlobalOptimSolution objects containing the distinct local solutions found during the run. The vector is sorted by objective function value; the first element is best (smallest value). The object contains:

X	Solution point returned by the local solver.
Fval	Objective function value at the solution.
Exitflag	Integer describing the result of the local solver run.
Output	Output structure returned by the local solver.
X0	Cell array of start points that led to the solution.

Examples   Use a default GlobalSearch object to solve the six-hump camel back problem (see "Run the Solver" on page 3-19):

```
gs = GlobalSearch;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(gs,problem);
```

Algorithm   A detailed description of the algorithm appears in "GlobalSearch Algorithm" on page 3-37. Ugray et al. [1] describes both the algorithm and the scatter-search method of generating trial points.

References   [1] Ugray, Zsolt, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. *Scatter Search and Local NLP Solvers: A Multistart Framework for Global Optimization.* INFORMS Journal on Computing, Vol. 19, No. 3, 2007, pp. 328–340.

See Also   GlobalSearch | createOptimProblem | GlobalOptimSolution

Tutorials   • "Run the Solver" on page 3-19

• "How to Optimize with GlobalSearch and MultiStart" on page 3-2

• "GlobalSearch and MultiStart Examples" on page 3-63

MultiStart.run

Purpose	Run local solver from multiple points
Syntax	*x* = run(*ms*,*problem*,*k*)
	x = run(*ms*,*problem*,*startpts*)
	[*x*,*fval*] = run(...)
	[*x*,*fval*,*exitflag*] = run(...)
	[*x*,*fval*,*exitflag*,*output*] = run(...)
	[*x*,*fval*,*exitflag*,*output*,*solutions*] = run(...)

Description   *x* = run(*ms*,*problem*,*k*) runs the *ms* MultiStart object on the optimization problem specified in *problem* for a total of *k* runs. *x* is the point where the lowest function value was found. For the lsqcurvefit and lsqnonlin solvers, MultiStart minimizes the sum of squares at *x*, also known as the residual.

x = run(*ms*,*problem*,*startpts*) runs the *ms* MultiStart object on the optimization problem specified in *problem* using the start points described in *startpts*.

[*x*,*fval*] = run(...) returns the lowest objective function value *fval* at *x*. For the lsqcurvefit and lsqnonlin solvers, *fval* contains the residual.

[*x*,*fval*,*exitflag*] = run(...) returns an exit flag describing the return condition.

[*x*,*fval*,*exitflag*,*output*] = run(...) returns an output structure containing statistics of the run.

[*x*,*fval*,*exitflag*,*output*,*solutions*] = run(...) returns a vector of solutions containing the distinct local minima found during the run.

Input Arguments   ms

    MultiStart object.

problem

Problem structure. Create `problem` with `createOptimProblem` or by exporting a problem structure from the Optimization Tool. `problem` must contain the following fields:

- `solver`
- `objective`
- `x0`
- `options` — Both `createOptimProblem` and the Optimization Tool always include an `options` field in the `problem` structure.

k

Number of start points to run. `MultiStart` generates `k - 1` start points using the same algorithm as `list` for a `RandomStartPointSet` object. `MultiStart` also uses the `x0` point from the `problem` structure.

startpts

A `CustomStartPointSet` or `RandomStartPointSet` object. `startpts` can also be a cell array of these objects.

Output Arguments

x

Point at which the objective function attained the lowest value during the run. For `lsqcurvefit` and `lsqnonlin`, the objective function is the sum of squares, also known as the residual.

fval

Smallest objective function value attained during the run. For `lsqcurvefit` and `lsqnonlin`, the objective function is the sum of squares, also known as the residual.

exitflag

Integer describing the return condition:

2	At least one local minimum found. Some runs of the local solver converged.
1	At least one local minimum found. All runs of the local solver converged.
0	No local minimum found. Local solver called at least once, and at least one local solver exceeded the MaxIter or MaxFunEvals tolerances.
-1	One or more local solver runs stopped by the local solver output or plot function.
-2	No feasible local minimum found.
-5	MaxTime limit exceeded.
-8	No solution found. All runs had local solver exit flag -2 or smaller, not all equal -2.
-10	Failures encountered in user-provided functions.

output

Structure containing statistics of the optimization. Fields in the structure:

funcCount	Number of function evaluations.
localSolverIncomplete	Number of local solver runs with 0 exit flag.
localSolverNoSolution	Number of local solver runs with negative exit flag.
localSolverSuccess	Number of local solver runs with positive exit flag.
localSolverTotal	Total number of local solver runs.
message	Exit message.

solutions

A vector of GlobalOptimSolution objects containing the distinct local solutions found during the run. The vector is sorted by

objective function value; the first element is best (smallest value). The object contains:

X	Solution point returned by the local solver.
Fval	Objective function value at the solution.
Exitflag	Integer describing the result of the local solver run.
Output	Output structure returned by the local solver.
X0	Cell array of start points that led to the solution.

Examples   Use a default MultiStart object to solve the six-hump camel back problem (see "Run the Solver" on page 3-19):

```
ms = MultiStart;
sixmin = @(x)(4*x(1)^2 - 2.1*x(1)^4 + x(1)^6/3 ...
    + x(1)*x(2) - 4*x(2)^2 + 4*x(2)^4);
problem = createOptimProblem('fmincon','x0',[-1,2],...
    'objective',sixmin,'lb',[-3,-3],'ub',[3,3]);
[xmin,fmin,flag,outpt,allmins] = run(ms,problem,30);
```

Algorithm   A detailed description of the algorithm appears in "MultiStart Algorithm" on page 3-41.

See Also   MultiStart | createOptimProblem | CustomStartPointSet | RandomStartPointSet | GlobalOptimSolution

Tutorials   • "Run the Solver" on page 3-19

• "How to Optimize with GlobalSearch and MultiStart" on page 3-2

• "GlobalSearch and MultiStart Examples" on page 3-63

RandomStartPointSet class

Purpose Random start points

Description Describes how to generate a set of pseudorandom points for use with `MultiStart`. A `RandomStartPointSet` object does not contain points. It contains parameters used in generating the points when `MultiStart` runs, or by the `list` method.

Construction *RS* = RandomStartPointSet constructs a default `RandomStartPointSet` object.

RS = RandomStartPointSet(*'PropertyName'*,*PropertyValue*,...) constructs the object using options, specified as property name and value pairs.

RS = RandomStartPointSet(*OLDRS*,*'PropertyName'*,*PropertyValue*,...) creates a copy of the *OLDRS* RandomStartPointSet object, with the named properties altered with the specified values.

Properties ArtificialBound

Absolute value of default bounds to use for unbounded problems (positive scalar).

If a component has no bounds, `list` uses a lower bound of -ArtificialBound, and an upper bound of ArtificialBound.

If a component has a lower bound `lb`, but no upper bound, `list` uses an upper bound of `lb + 2*ArtificialBound`. Similarly, if a component has an upper bound `ub`, but no lower bound, `list` uses a lower bound of `ub - 2*ArtificialBound`.

Default: 1000

NumStartPoints

Number of start points to generate (positive integer)

Default: 10

Methods

list	Generate start points

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Create a `RandomStartPointSet` object for 40 points, and use `list` to generate a point matrix for a seven-dimensional problem:

```
rs = RandomStartPointSet('NumStartPoints',40); % 40 points
problem = createOptimProblem('fminunc','x0',ones(7,1),...
    'objective',@rosenbrock);
ptmatrix = list(rs,problem); % 'list' generates the matrix
```

See Also

MultiStart | CustomStartPointSet | list

Tutorials

- "RandomStartPointSet Object for Start Points" on page 3-17

How To

- Class Attributes

- Property Attributes

saoptimget

Purpose   Values of simulated annealing 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 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 Options" on
                 page 9-53.

                 saoptimset, simulannealbnd

Purpose   Create simulated annealing options structure

Syntax   ```
saoptimset
options = saoptimset
options = saoptimset('param1',value1,'param2',value2,...)
options = saoptimset(oldopts,'param1',value1,...)
options = saoptimset(oldopts,newopts)
options = saoptimset('simulannealbnd')
```

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 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('simulannealbnd') creates an options structure with all the parameter names and default values relevant to 'simulannealbnd'. For example,

```
saoptimset('simulannealbnd')

ans =
```

```
           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'
```

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 |{@acceptancesa}
AnnealingFcn	Handle to the function the algorithm uses to generate new points	Function handle | @annealingboltz | {@annealingfast}
DataType	Type of decision variable	'custom' | {'double'}

Option	Description	Values
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 scalar | `{100}`
MaxFunEvals	Maximum number of objective function evaluations allowed	Positive integer | `{3000*numberOfVariables}`
MaxIter	Maximum number of iterations allowed	Positive integer | `{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 | `{[]}`
PlotFcns	Plot function(s) called during iterations	Function handle or cell array of function handles | `@saplotbestf` | `@saplotbestx` | `@saplotf` | `@saplotstopping` | `@saplottemperature` | `{[]}`

saoptimset

Option	Description	Values
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 Options" on page 9-53.

saoptimget, 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
rngstate	Optional field to reset the state of the random number generator

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 documentation.

`[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.

- rngstate — State of the MATLAB random number generator, just before the algorithm started. You can use the values in rngstate to reproduce the output of simulannealbnd. See "Reproducing Your Results" on page 6-17.

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.
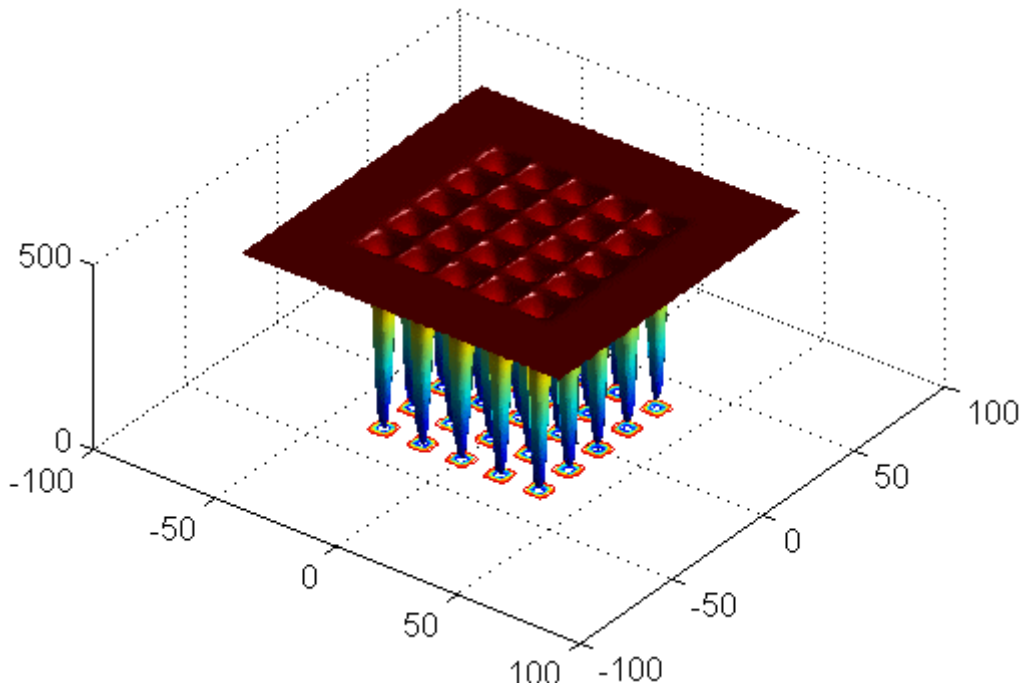

```
x0 = [0 0];
[x,fval] = simulannealbnd(@dejong5fcn,x0)
```

```
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.

simulannealbnd
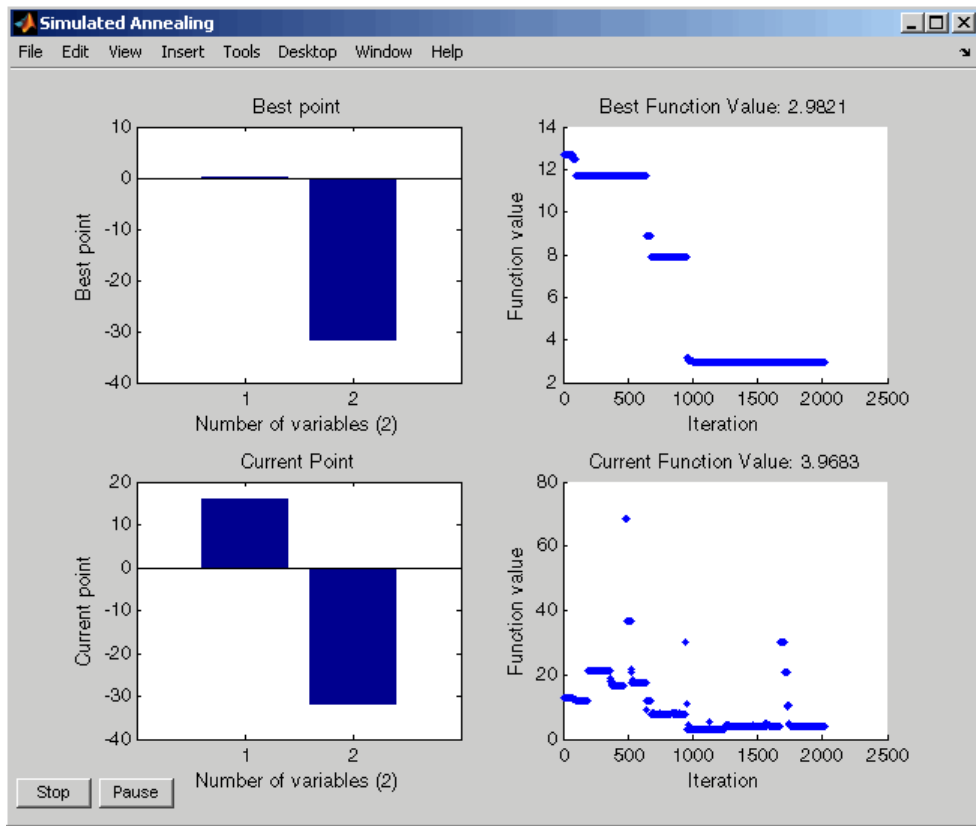
See Also   ga, patternsearch, saoptimget, saoptimset

Examples

Use this list to find examples in the documentation.

GlobalSearch and MultiStart

Pattern Search

Genetic Algorithm

Simulated Annealing

Index