Optimization with the Genetic Algorithm/Direct Search Toolbox

Ed Hall

edhall@virginia.edu

Research Computing Support Center

Wilson Hall, Room 244

University of Virginia

Phone: 243-8800

Email: Res-Consult@Virginia.EDU

URL: www.itc.virginia.edu/researchers/services.html

Outline of Talk

- Optimization Overview
- Gradient-Based Optimization
- Direct Search Method
- Genetic Algorithm
- Conclusions

Optimization Overview

Given function $f : R^n \to R$, and set $S \subseteq R^n$, find $x^* \in S$ such that $f(x^*) \leq f(x)$ for all $x \in S$.

x^* called the *minimizer* or *minimum* if f.

Suffices to consider only minimization, since maximum of f is minimum of $-f$.

Objective function f usually assumed differentiable, may be linear or nonlinear.

Constraint set S defined by system of equations and equalities that may be linear of nonlinear.

Points $x \in S$ called *feasible* points.

if $S = R^n$, problem is *unconstrained*.

Optimization Problems

General optimization problem:

min $f(x)$ subject to $g(x) = 0$ and $h(x) \leq 0$,

where $f : R^n \to R$, $g : R^n \to R^m$, $h : R^n \to R^p$.

Linear programming: f, g, and h all linear.

Nonlinear programming: nonlinear objective or nonlinear constraints, or both.

In this talk, we'll consider the following *nonlinear programming* problems:

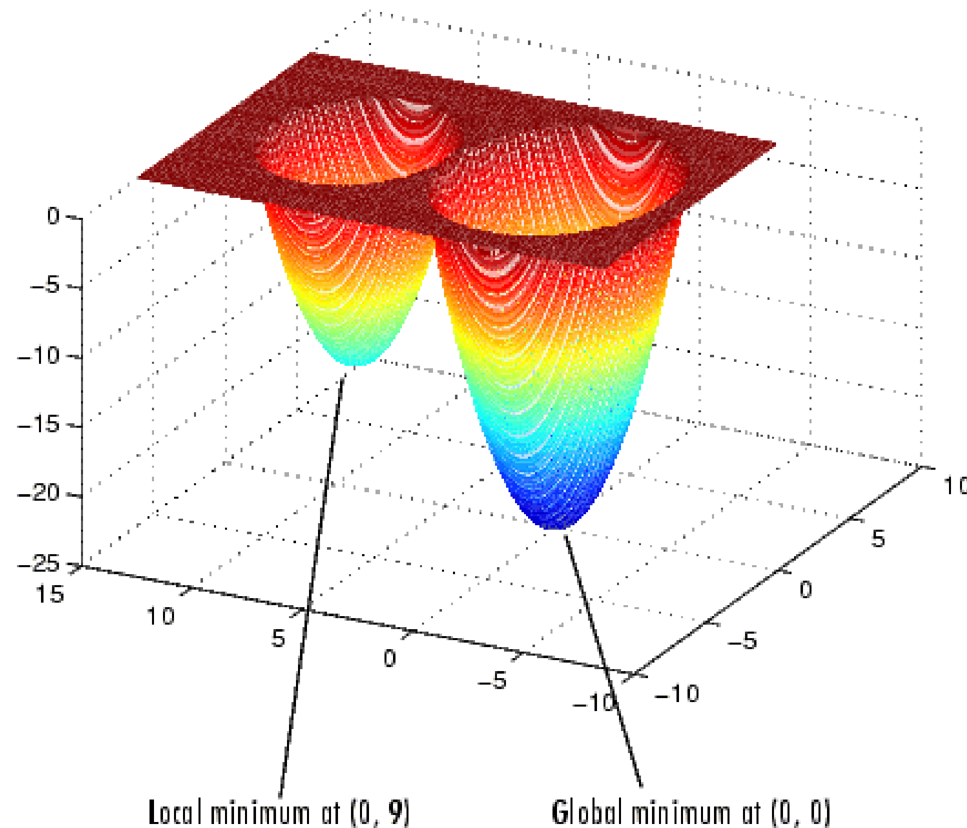
Unconstrained: min $f(x)$ for all $x \in R^n$

Linear Constraints: min $f(x)$ for all x such that $Ax \leq b$ and $A_{eq}x = b_{eq}$ where A and A_{eq} are matrices and b and b_{eq} are vectors.

Local vs. Global Optimization

$x^* \in S$ is a *global minimum* if $f(x^*) \leq f(x)$ for all $x \in S$.

$x^* \in S$ is a *local minimum* if $f(x^*) \leq f(x)$ for all feasible x in some neighborhood of x^*.



Global Optimization

Finding, or even verifying, global minimum is difficult, in general

Most optimization methods designed to find local minimum, which may or may not be global minimum

If global minimum desired, can try several widely separated starting points and see if all produce same result

For some problems, such as linear programming, global optimization tractable

Gradient-Based Optimization

First-Order Optimality Condition

For a function f of n variables, find *critical point* (or *stationary point*), i.e. the solution of the nonlinear system,

$$\nabla f = 0,$$

where ∇f is the *gradient* vector of f, whose ith component is $\partial f(x)/\partial x_i$.

For a continuosly differentialbe $f : S \subseteq R^n \rightarrow R$, any interior point x^* of S at which f has a local minimum must be a critical point if f.

But not critical points are minima: can also be a maximum or saddle point.

Gradient-Based Optimization

Steepest Descent

Let $f : R^n \to R$ be real-valued function of n real variables. At any point x where gradient vector is nonzero, negative gradient, $-\nabla f(x)$, points downhill toward lower values of function f.

In fact, $-\nabla f(x)$ is locally direction of steepest descent: function decreases more rapidly along direction of negative gradient than along any other

Steepest descent method: starting from initial guess x0, successive approximate solutions given by

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

where α_k is *line* search parameter that determines how far to go in given direction

Similarly, gradient-based optimization methods use $\nabla f(x)$ in determining search direction.

Direct Search Algorithms

Direct search is a method for solving optimization problems that does not require any information about the gradient of the objective function.

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 methods 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.**

We will focus on a special class of direct search algorithms called pattern search algorithms.

Pattern Search

A pattern search algorithm computes a sequence of points that get closer to the optimal point.

1. 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.
2. The algorithm forms the mesh by adding the current point to a scalar multiple of a fixed set of vectors called a pattern.
3. If the algorithm finds a point in the mesh that improves the objective function at the current point, the new point becomes the current point at the next step of the algorithm.

Performing a Pattern Search

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

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

- @objfun is a handle to the objective function.
- x0 is the starting point for the pattern search.
- `fval` – Final value of the objective function
- `x` – Point at which the final value is attained

Alternatively, the pattern search can be intiated from a GUI interface by entering

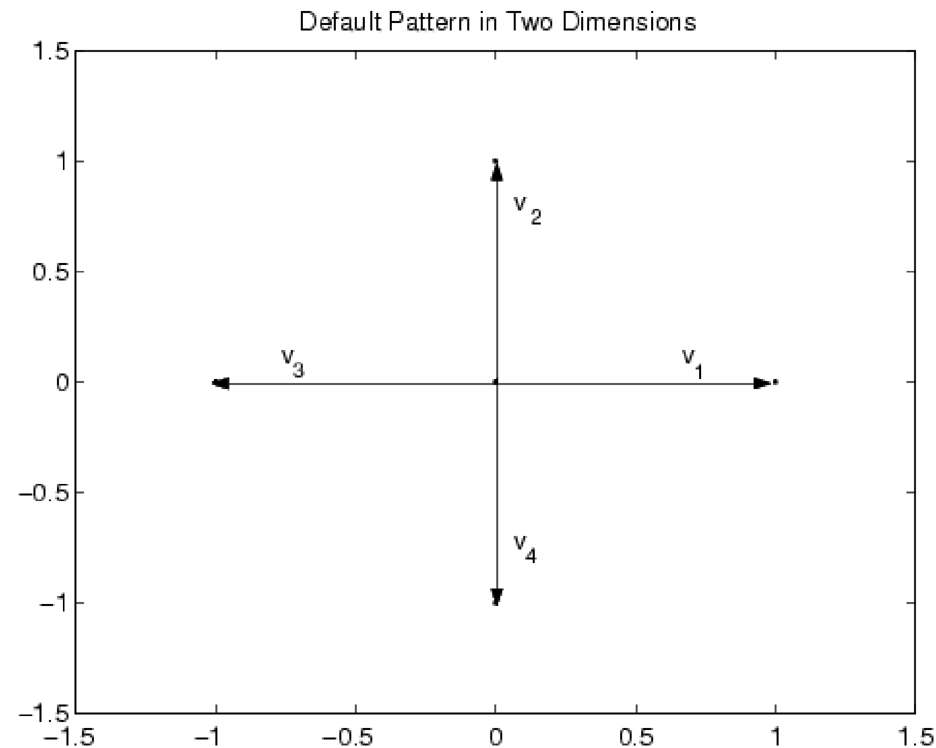
```
psearchtool
```

Patterns

A pattern is a collection of vectors that the algorithm uses to determine which points to search at each iteration.

For example, if there are two independent variables in the optimization problem, the default pattern consists of the following vectors.

```
v1 = [1 0] v2 = [0 1] v3 = [-1 0] v4 = [0 -1]
```



Meshes

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

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

For example, suppose that the current point is `[1.6 3.4]` and the mesh size if 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*.

Pattern Search Polling

At each step, the algorithm polls the points in the current mesh by computing their objective function values.

By default 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. The poll is then called *successful* and that point becomes the current point at the next iteration. If you set **Complete poll** to **On**, the algorithm computes the objective function values at all mesh points.

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

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.

After an *unsuccessful poll*, the algorithm multiplies the current mesh size by 0.5, the default value of **Mesh Contraction factor**. The algorithm then polls with a smaller mesh size.

Displaying the Results at Each Iteration

You can display the results of the pattern search at each iteration by setting **Level of display** to **Iterative** in **Display to command window** options of `psearchtool`.

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

Iter	f-count	f(x)	MeshSize	Method
0	1	4.645	1	Start iterations
1	4	4.334	2	Successful Poll
2	7	0.4763	4	Successful Poll
3	11	0.4763	2	Refine Mesh
4	15	0.4763	1	Refine Mesh
5	19	-0.9237	2	Successful Poll
6	23	-0.9237	1	Refine Mesh

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

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 search options are:

- Positive basis Np1
- Positive basis 2N
- Genetic Algorithm
- Latin hypercube
- Nelder-Mead
- Custom

Pattern Search Stopping Conditions

The algorithm stops when any of the following conditions occurs:

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

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

Linearly Constrained Minimization

The pattern search function can solve a linearly constrained minimization problem of the form

min $f(x)$ for all x such that $Ax \leq b$, $A_{eq}x = b_{eq}$, and $lb \leq x \leq ub$

with the command line syntax

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

- `A` and `b` are the matrix and vector in the inequality constraint.
- `Aeq` and `beq` are the matrix and vector in the equality constraint.
- `lb` and `ub` are the lower bound and upper bound on `x`.
- `options` is a structure containing options for the pattern search created with the `psoptimset` command. If you do not pass in this argument, patternsearch uses its default options.
- Pass empty matrices for verb=A=, `b`, `Aeq`, `beq`, `lb`, `ub`, and `options` to use default values.

Genetic Algorithm

The genetic algorithm is a method for solving optimization problems that is based on natural selection, the process that drives biological evolution.

The genetic algorithm repeatedly modifies a population of individual solutions.

At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them produce the children for the next generation.

Over successive generations, the population evolves toward an optimal solution.

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

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

Using the Genetic Algorithm

To use the genentic algorithm on an unconstrained problem at the Matlab command line, you call the function `ga` with the syntax

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

- `@fitness` 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 created with the `gaoptimset` command. If you do not pass in this argument, ga uses its default options.
- `fval` – Final value of the fitness function
- `x` – Point at which the final value is attained

Alternatively, the Genetic Algorithm Tool can be invoked by entering the command

```
gatool
```

Genetic Algorithm Terminology

Fitness Functions

The *fitness function* (objective function) is the function you want to optimize.

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. 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.

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.

Genetic Algorithm Terminology

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

Fitness Values and Best Fitness Values

The fitness value of an individual is the value of the fitness function for that individual. Because the toolbox finds the minimum of the fitness function, the best fitness value for a population is the smallest fitness value for any individual in the population.

Parents and Children

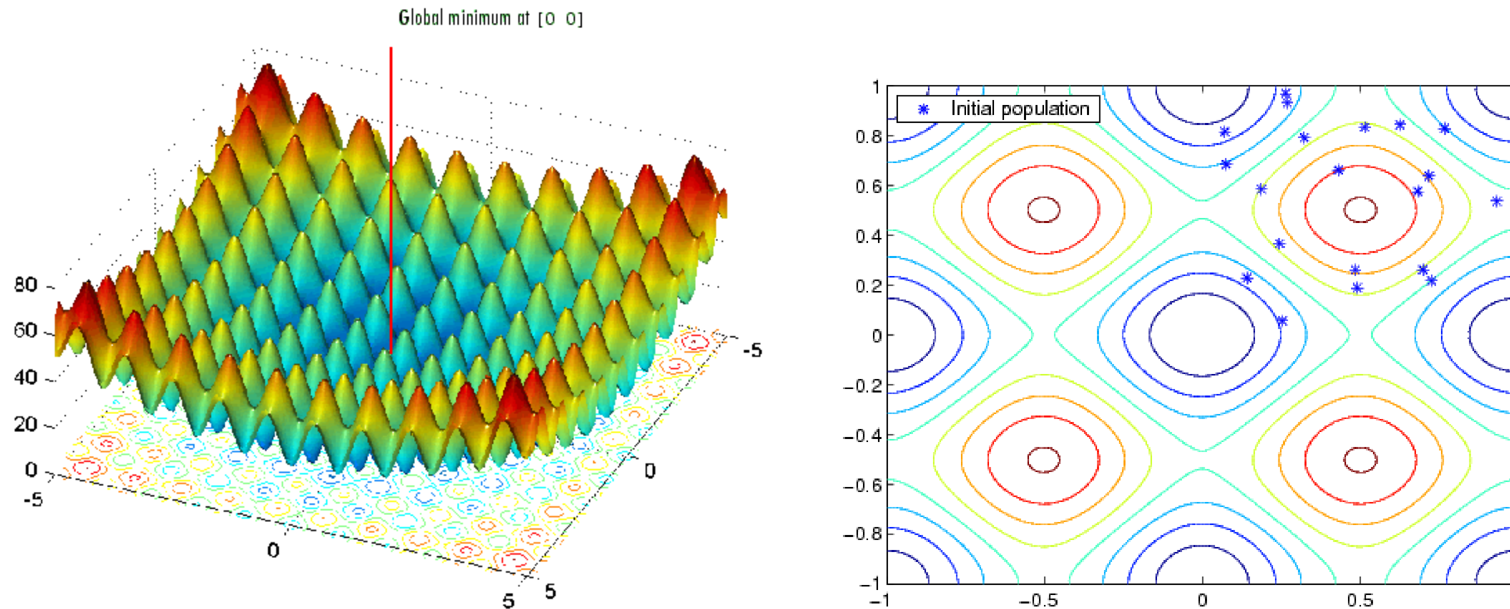
To create the next generation, the genetic algorithm selects certain individuals in the current population, called parents, and uses them to create individuals in the next generation, called children. Typically, the algorithm is more likely to select parents that have better fitness values.

Genetic Algorithm Outline

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

Initial Population

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



The initial population contains 20 individuals by default. All the individuals in the initial population have genes that lie between 0 and 1, because the default value of Initial range in the Population options is [0;1].

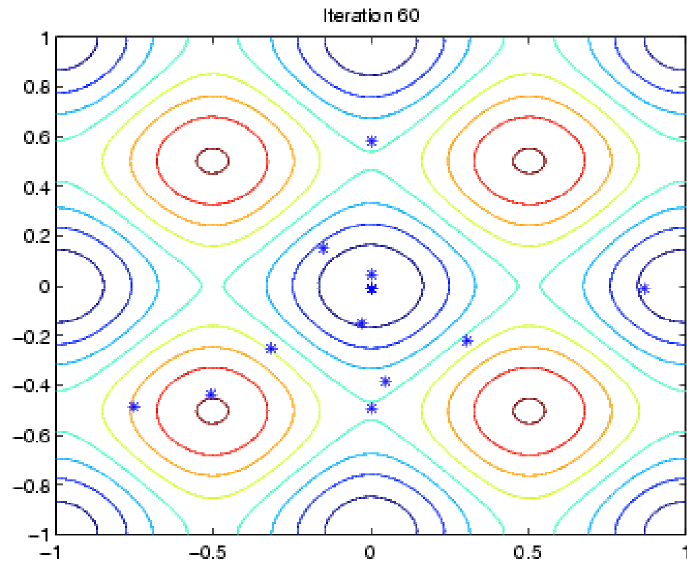
Creating the Next Generation

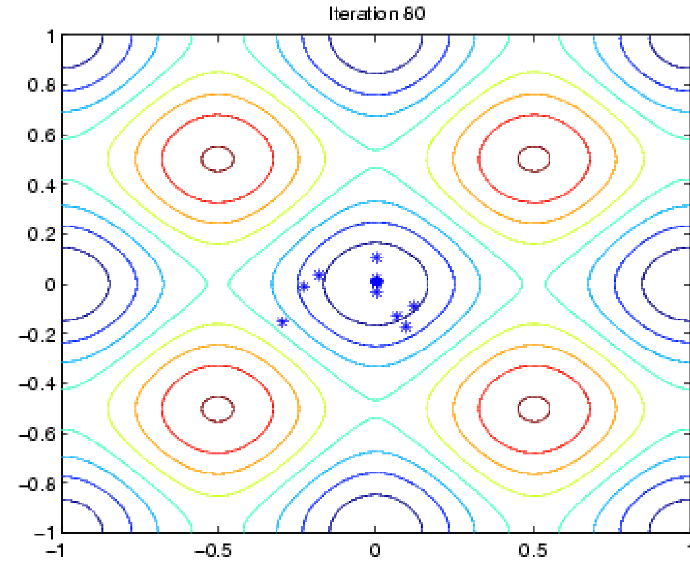
The algorithm selects individuals in the current population, called parents, who contribute their genes (the entries of their vectors) to their children. The algorithm most likely selects individuals that have better fitness values as parents.

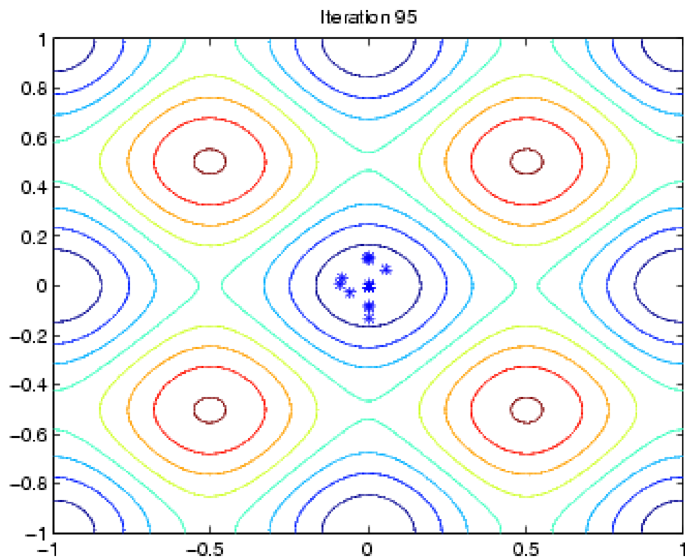
The genetic algorithm creates three types of children for the next generation:

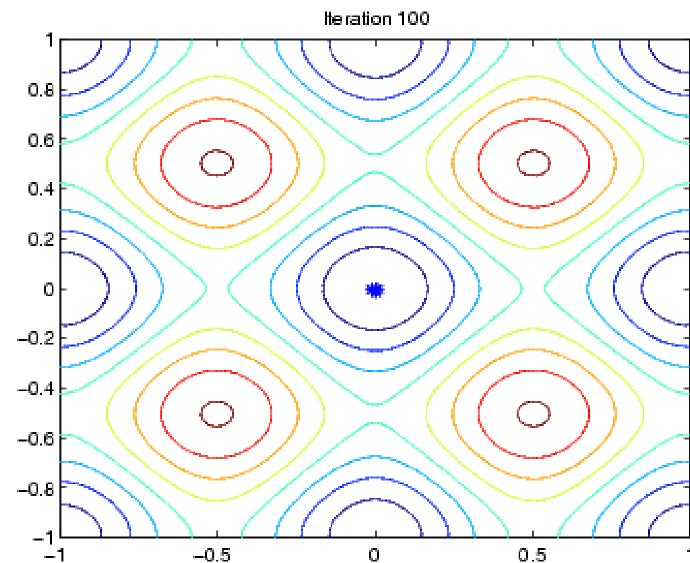
- *Elite children* are the individuals in the current generation with the best fitness values.
- *Crossover children* are created by combining the vectors of a pair of parents. At each coordinate of the child vector, the default crossover function randomly selects an entry, or gene, at the same coordinate from one of the two parents and assigns it to the child.
- *Mutation children* are created by introducing random changes, or mutations, to a single parent. By default, the algorithm adds a random vector from a Gaussian distribution to the parent.

Plots of Later Generations









Genetic Algorithm Stopping Conditions

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

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

Using a Hybrid Function

The genetic algorithm can reach the region near an optimum point relatively quickly, but it can take many function evaluations to achieve convergence.

A commonly used technique is to run genetic algorithm for a small number of generations to get near an optimum point. Then the solution from genetic algorithm is used as an initial point for another optimization solver that is faster and more efficient for local search.

You can specify a hybrid function in **Hybrid function** options, e.g. `fminunc`.

Usually close the the minimum the objective function is smooth enough that a standard optimization algorithm can then be used.

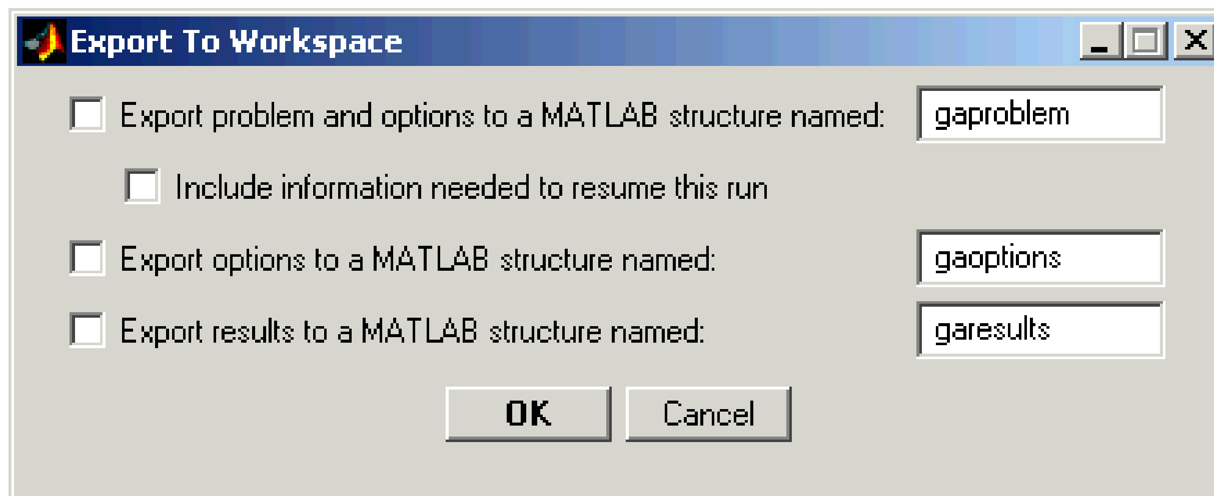
To see a demo of using a hybrid function, run `hybriddemo`.

Exporting Options and Problems

You can export options and problem structures from the Genetic Algorithm (or Pattern Search) Tool to the MATLAB workspace, and then import them back into the tool at a later time.

You can also export the options structure and use it with the genetic algorithm function `ga` (or `patternsearch`) at the command line.

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



Parameterizing Objective Functions

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

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

for different values of a, b, and c. Because `patternsearch` and `ga` accept objective or fitness functions that depend only on x, you must provide the additional parameters a, b, and c to the function before calling `patternsearch` or `ga`.

The following sections describe two ways to do this:

- Using Anonymous Functions
- Using a Nested Function

Using Anonymous Functions

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

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

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

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

Using a Nested Function

Instead of parameterizing the objective function as an anonymous function, you can write a single M-file that

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

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

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

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

Vectorizing the Objective Function

These algorithms usually runs faster if you vectorize the objective function, so that one call computes the fitness for all individuals in the current population, or values of all points in the current mesh at once. To vectorize the fitness function,

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

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

  write the M-file using the following code:

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

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

Using the GA/DS Toolbox

Conclusions

- Experiment with the algorithm options to try and find the global minimum.
- Use the graphical interface for the pattern search and genetic algorithm (`psearchtool` and `gatool`) to easily experiment with different options and then save the options structure.
- Perform extended optimizations, e.g. using parameterized objective functions, using the command line interfaces (`patternsearch` and `ga`).
- Vectorization the objective function, if possible.

GA/DS Toolbox References

The Genetic Algorithm and Direct Search Toolbox Documentation.
www.mathworks.com/access/helpdesk/help/toolbox/gads/

Scientific Computing: An Introductory Survey by Michael T. Heath.
Chapter 6: Optimization
www.cse.uiuc.edu/heath/scicomp/

Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods by T. Kolda, R. Lewis. and V. Torczon.
SIAM Review, Vol. 45, No. 3, pp. 385-492.

Genetic Algorithms in Search, Optimzation & Machine Learning by David E. Goldberg.
Addison-Wesley, 1989.

Upcoming Talks at the RCSC

- April 20, 3:30 PM: "Statistical Shape Analysis" by Kathy Gerber.
- April 27, 3:30 PM: "Objected Oriented Programming with Fortran 95 (FTN95)" by Katherine Holcomb.

Further information on these talks can be found at the URL
www.itc.virginia.edu/research/news/newsletterMar05.html#colloquia