Optimization Toolbox

For Use with MATLAB®

Thomas Coleman
Mary Ann Branch
Andrew Grace

Computation

Visualization

Programming

The MATH WORKS Inc.

User's Guide

Version 2

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Optimization Toolbox User's Guide

Printing History:	November 1990	First printing
	December 1996	Second printing for MATLAB 5
	January 1999	Third printing Revised for Version 2 (Release 11)

Contents

Before You Begin

1 Tutorial

2 Introduction to Algorithms

4 Reference

Before You Begin

What Is the Optimization Toolbox?

The Optimization Toolbox is a collection of functions that extend the capability of the MATLAB® numeric computing environment. The toolbox includes routines for many types of optimization including:

- Unconstrained nonlinear minimization
- Constrained nonlinear minimization, including goal attainment problems, minimax problems, and semi-infinite minimization problems
- Quadratic and linear programming
- Nonlinear least squares and curve-fitting
- Nonlinear system of equation solving
- Constrained linear least squares

Specialized algorithms for large-scale (sparse) problems are also available (see "Large-Scale Algorithms" in the next section "New Features in Version 2").

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

```
type function_name
```

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

New Features in Version 2

Version 2 of the Optimization Toolbox offers a number of advances over previous versions.

Large-Scale Algorithms

The focus of this version is new algorithms for solving large-scale problems, including

- Linear programming
- Nonlinear least squares with bound constraints
- Nonlinear system of equation solving
- Unconstrained nonlinear minimization
- Nonlinear minimization with bound constraints
- Nonlinear minimization with linear equalities
- Quadratic problems with bound constraints
- Quadratic problems with linear equalities
- Linear least squares with bound constraints

The new large-scale algorithms have been incorporated into the toolbox functions. The new functionality improves the ability of the toolbox to solve large sparse problems.

Function Names and Calling Syntax

To accommodate this new functionality, many of the function names and calling sequences have changed. Some of the improvements include

- Command line syntax has changed:
  - Equality constraints and inequality constraints are now supplied as separate input arguments.
  - Linear constraints are supplied as separate arguments from the nonlinear constraint function.
  - The gradient of the objective is computed in the same function as the objective, rather than in a separate function, in order to provide more efficient computation (because the gradient and objective often share

  similar computations). Similarly, the gradient of the nonlinear constraints is computed by the (now separate) nonlinear constraint function.
  - The Hessian matrix can be provided by the objective function when using the large-scale algorithms.
- Optimization parameters are now contained in a structure, with functions to create, change, and retrieve values.
- Each function returns an exit flag that denotes the termination state.

For more information on how to convert your old syntax to the new function calling sequences, see "Converting Your Code to Version 2.0 Syntax" in Chapter 1.

How to Use this Manual

This manual has three main parts:

- Chapter 1 provides a tutorial for solving different optimization problems, including a special section that highlights large-scale problems. This chapter also provides information on how to use the toolbox functions in conjunction with Simulink using multiobjective optimization. Other sections include information about changing default parameters and using inline objects.
- Chapters 2 and 3 describe the algorithms in the optimization functions. Chapter 2 describes the problem formulations and the algorithms for the medium-scale algorithms. Chapter 3 focuses on the large-scale algorithms.
- Chapter 4 provides a detailed reference description of each toolbox function. Reference descriptions include the function's syntax, a description of the different calling sequences available, and detailed information about arguments to the function, including relevant optimization options parameters. Reference descriptions may also include examples, a summary of the function's algorithms, and references to additional reading material.

Installing the Toolbox

To determine if the Optimization Toolbox is installed on your system, type this command at the MATLAB prompt:

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

If the Optimization Toolbox is not installed, check the *Installation Guide* for instructions on how to install it.

Typographical Conventions

To Indicate	This Guide Uses	Example
Example code	`Monospace type`	To assign the value 5 to A, enter: `A = 5`

MATLAB output	`Monospace type`	MATLAB responds with
		`A =`
		`     5`

Function names	`Monospace type`	The `cos` function finds the cosine of each array element.
Function syntax	`Monospace type` for text that must appear as shown.	The `fminbnd` function uses the syntax `x= fminbnd('sin',3,4)`
Mathematical expressions	Variables in *italics*. Functions, operators, and constants in standard type.	This vector represents the polynomial $p = x^2 + 2x + 3$
New terms	*Italics*	An *array* is an ordered collection of information.

Matrix, Vector, and Scalar Notation

Upper-case letters such as A are used to denote matrices. Lower-case letters such as x are used to denote vectors, except where noted that it is a scalar. For functions, the notation differs slightly to follow the usual conventions in optimization. For vector functions, we use an upper-case letter such as F in $F(x)$. A function that returns a scalar value is denoted with a lower-case letter such as f in $f(x)$.

Tutorial

Introduction

Optimization concerns the minimization or maximization of functions. The Optimization Toolbox consists of functions that perform minimization (or maximization) on general nonlinear functions. Functions for nonlinear equation solving and least-squares (data-fitting) problems are also provided.

The tables below show the functions available for minimization, equation solving, and solving least squares or data fitting problems.

Table 1-1: Minimization

Type	Notation	Function
Scalar Minimization	$\min_{a} f(a)$ such that $a_1 < a < a_2$	`fminbnd`
Unconstrained Minimization	$\min_{x} f(x)$	`fminunc`, `fminsearch`
Linear Programming	$\min_{x} f^T x$ such that $A \cdot x \le b,\quad Aeq \cdot x = beq,\quad l \le x \le u$	`linprog`
Quadratic Programming	$\min_{x} \frac{1}{2}x^T Hx + f^T x$ such that $A \cdot x \le b,\quad Aeq \cdot x = beq,\quad l \le x \le u$	`quadprog`
Constrained Minimization	$\min_{x} f(x)$ such that $c(x) \le 0,\ ceq(x) = 0$ $A \cdot x \le b,\quad Aeq \cdot x = beq,\quad l \le x \le u$	`fmincon`
Goal Attainment	$\min_{x,\gamma} \gamma$ such that $F(x) - w\gamma \le$ goal $c(x) \le 0,\ ceq(x) = 0,$ $A \cdot x \le b,\quad Aeq \cdot x = beq,\quad l \le x \le u$	`fgoalattain`

Table 1-1: Minimization (Continued)

Type	Notation	Function
Minimax	$\min_{x} \max_{\{F_i\}} \{F_i(x)\}$ such that $c(x) \le 0,\ ceq(x) = 0,$ $A \cdot x \le b,\ Aeq \cdot x = beq,\ l \le x \le u$	`fminimax`
Semi-infinite Minimization	$\min_{x} f(x)$ such that $K(x, w) \le 0$ for all $w,$ $c(x) \le 0,\ ceq(x) = 0,$ $A \cdot x \le b,\ Aeq \cdot x = beq,\ l \le x \le u$	`fseminf`

Table 1-2: Equation Solving

Type	Notation	Function
Linear Equations	$C \cdot x = d$, n equations, n variables	`\ (slash)`
Nonlinear Equation of One Variable	$f(a) = 0$	`fzero`
Nonlinear Equations	$F(x) = 0$, n equations, n variables	`fsolve`

Table 1-3: Least-Squares (Curve Fitting)

Type	Notation	Function
Linear Least Squares	$\min_{x} \lVert C \cdot x - d \rVert_2^2$, m equations, n variables	`\ (slash)`
Nonnegative Linear Least Squares	$\min_{x} \lVert C \cdot x - d \rVert_2^2$ such that $x \ge 0$	`lsqnonneg`

Table 1-3: Least-Squares (Curve Fitting) (Continued)

Type	Notation	Function
Constrained Linear Least Squares	$\min_x \lVert C \cdot x - d \rVert_2^2$ such that $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u$	`lsqlin`
Nonlinear Least Squares	$\min_x \frac{1}{2}\lVert F(x) \rVert_2^2 = \frac{1}{2}\sum_i F_i(x)^2$ such that $l \le x \le u$	`lsqnonlin`
Nonlinear Curve Fitting	$\min_x \frac{1}{2}\lVert F(x, xdata) - ydata \rVert_2^2$ such that $l \le x \le u$	`lsqcurvefit`

Overview

Most of these optimization routines require the definition of an M-file containing the function to be minimized. Alternatively, an inline object created from a MATLAB expression can be used. Maximization is achieved by supplying the routines with `–f`, where `f` is the function being optimized.

Optimization options passed to the routines change optimization parameters. Default optimization parameters are used extensively but can be changed through an `options` structure.

Gradients are calculated using an adaptive finite-difference method unless they are supplied in a function. Parameters can be passed directly to functions, avoiding the need for global variables.

This *User's Guide* separates "medium-scale" algorithms from "large-scale" algorithms. Medium-scale is not a standard term and is used here only to differentiate these algorithms from the large-scale algorithms, which are designed to handle large-scale problems efficiently.

Medium-Scale Algorithms

The Optimization Toolbox routines offer a choice of algorithms and line search strategies. The principal algorithms for unconstrained minimization are the Nelder-Mead simplex search method and the BFGS quasi-Newton method. For constrained minimization, minimax, goal attainment, and semi-infinite optimization, variations of Sequential Quadratic Programming are used.

Nonlinear least squares problems use the Gauss-Newton and Levenberg-Marquardt methods.

A choice of line search strategy is given for unconstrained minimization and nonlinear least squares problems. The line search strategies use safeguarded cubic and quadratic interpolation and extrapolation methods.

Large-Scale Algorithms

All the large-scale algorithms, except linear programming, are trust-region methods. Bound constrained problems are solved using reflective Newton methods. Equality constrained problems are solved using a projective preconditioned conjugate gradient iteration. You can use sparse iterative solvers or sparse direct solvers in solving the linear systems to determine the current step. Some choice of preconditioning in the iterative solvers is also available.

The linear programming method is a variant of Mehrotra's predictor-corrector algorithm, a primal-dual interior-point method.

Medium-Scale Examples

The Optimization Toolbox presents *medium-scale* algorithms through a tutorial. The first part of this tutorial (through the “Equality Constrained Example”) follows the first demonstration *Tutorial Walk Through* in the M-file `optdemo`. The examples in the manual differ in that M-file functions were written for the objective functions, whereas in the online demonstration, inline objects were used for some functions.

The tutorial discusses the functions `fminunc` and `fmincon` in detail. The other optimization routines `fgoalattain`, `fminimax`, `lsqnonlin`, `fsolve`, and `fseminf` are used in a nearly identical manner, with differences only in the problem formulation and the termination criteria. The next section discusses multiobjective optimization and gives several examples using `lsqnonlin`, `fminimax`, and `fgoalattain` including how Simulink can be used in conjunction with the toolbox.

Unconstrained Example

Consider the problem of finding a set of values $[x_1, x_2]$ that solves

$$\underset{x}{\text{minimize}} \quad f(x) \quad = \quad e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \qquad \textbf{(1-1)}$$

To solve this two-dimensional problem, write an M-file that returns the function value. Then, invoke the unconstrained minimization routine `fminunc`.

Step 1: Write an M-file objfun.m

```
function f = objfun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
```

Step 2: Invoke one of the unconstrained optimization routines

```
x0 = [–1,1];    % Starting guess
options = optimset('LargeScale','off');
[x,fval,exitflag,output] = fminunc('objfun',x0,options);
```

After 40 function evaluations, this produces the solution:

```
x =
       0.5000   –1.0000
```

The function at the solution `x` is returned in `fval`:

```
fval =
      1.3030e-10
```

The `exitflag` tells if the algorithm converged. An `exitflag` > 0 means a local minimum was found:

```
exitflag =
     1
```

The `output` structure gives more details about the optimization. For `fminunc`, it includes the number of iterations in `iterations`, the number of function evaluations in `funcCount`, the final step-size in `stepsize`, a measure of first-order optimality (which in this unconstrained case is the infinity norm of the gradient at the solution) in `firstorderopt`, and the type of algorithm used in `algorithm`:

```
output =
       iterations: 7
        funcCount: 40
         stepsize: 1
    firstorderopt: 9.2801e-004
        algorithm: 'medium-scale: Quasi-Newton line search'
```

When there exists more than one local minimum, the initial guess for the vector $[x_1, x_2]$ affects both the number of function evaluations and the value of the solution point. In the example above, `x0` is initialized to `[-1,1]`.

The variable `options` can be passed to `fminunc` to change characteristics of the optimization algorithm, as in

```
x = fminunc('objfun',x0,options);
```

`options` is a structure that contains values for termination tolerances and algorithm choices. An `options` structure can be created using the `optimset` function

```
options = optimset('LargeScale','off');
```

In this example we have turned off the default selection of the large-scale algorithm and so the medium-scale algorithm is used. Other options include controlling the amount of command line display during the optimization iteration, the tolerances for the termination criteria, if a user-supplied gradient or Jacobian is to be used, and the maximum number of iterations or function evaluations. See the *References* chapter pages for `optimset` and the individual optimization functions, and Table 4-3 for more options and information.

Nonlinear Inequality Constrained Example

If inequality constraints are added to Eq. 1-1, the resulting problem may be solved by the `fmincon` function. For example, find x that solves

$$\underset{x}{\text{minimize}} \quad f(x) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

$$\text{subject to the constraints:} \qquad \begin{aligned} x_1x_2 - x_1 - x_2 &\le -1.5 \\ x_1x_2 &\ge -10 \end{aligned} \tag{1-2}$$

Since neither of the constraints is linear, you cannot pass the constraints to `fmincon` at the command line; instead you can create a second M-file `confun.m` that returns the value at both constraints at the current `x` in a vector `c`. The constrained optimizer, `fmincon`, is then invoked. Because `fmincon` expects the constraints to be written in the form $c(x) \le 0$, you must rewrite your constraints in the form

$$\begin{aligned} x_1x_2 - x_1 - x_2 + 1.5 &\le 0 \\ -x_1x_2 - 10 &\le 0 \end{aligned} \tag{1-3}$$

Step 1: Write an M-file confun.m for the constraints

```
function [c, ceq] = confun(x)
% nonlinear inequality constraints
c = [1.5 + x(1)*x(2) — x(1) — x(2);
    —x(1)*x(2) — 10];
% nonlinear equality constraints
ceq = [];
```

Step 2: Invoke constrained optimization routine:

```
x0 = [-1,1];   % Make a starting guess at the solution
options = optimset('LargeScale','off');
[x, fval] = ...
fmincon('objfun',x0,[],[],[],[],[],[],'confun',options)
```

After 38 function calls, the solution `x` produced with function value `fval` is

```
x =
   -9.5474  1.0474
fval =
    0.0236
```

We can evaluate the constraints at the solution

```
[c,ceq] = confun(x)
c=
    1.0e-15 *
   -0.8882
    0
ceq =
     []
```

Note that both constraint values are less than or equal to zero, that is, `x` satisfies $c(x) \le 0$.

Constrained Example with Bounds

The variables in `x` can be restricted to certain limits by specifying simple bound constraints to the constrained optimizer function. For `fmincon`, the command

```
x = fmincon('objfun',x0,[],[],[],[],lb,ub,'confun',options);
```

limits `x` to be within the range `lb <= x <= ub`.

To restrict x in Eq. 1-2 to be greater than zero (i.e., $x_1 \geq 0$, $x_2 \geq 0$), use the commands:

```
x0 = [−1,1];        % Make a starting guess at the solution
lb = [0,0];         % Set lower bounds
ub = [ ];           % No upper bounds
options = optimset('LargeScale','off');
[x,fval = ...
      fmincon('objfun',x0,[],[],[],[],lb,ub,'confun',options)
[c, ceq] = confun(x)
```

Note that to pass in the lower bounds as the seventh argument to `fmincon`, you must specify values for the third through sixth arguments. In this example, we specified `[]` for these arguments since we have no linear inequalities or linear equalities.

After 13 function evaluations, the solution produced is

```
x =
       0   1.5000
fval =
       8.5000
c =
     0
   −10
ceq =
     []
```

When `lb` or `ub` contains fewer elements than `x`, only the first corresponding elements in `x` are bounded. Alternatively, if only some of the variables are bounded, then use `−inf` in `lb` for unbounded below variables and `inf` in `ub` for unbounded above variables. For example,

```
lb = [−inf 0];
ub = [10 inf];
```

bounds $x_1 \leq 10$, $0 \leq x_2$ (x_1 has no lower bound and x_2 has no upper bound). Using `inf` and `−inf` give better numerical results than using a very large positive number or a very large negative number to imply lack of bounds.

Note that the number of function evaluations to find the solution is reduced since we further restricted the search space. Fewer function evaluations are usually taken when a problem has more constraints and bound limitations because the optimization makes better decisions regarding step-size and regions of feasibility than in the unconstrained case. It is, therefore, good practice to bound and constrain problems, where possible, to promote fast convergence to a solution.

Constrained Example with Gradients

Ordinarily the medium-scale minimization routines use numerical gradients calculated by finite-difference approximation. This procedure systematically perturbs each of the variables in order to calculate function and constraint partial derivatives. Alternatively, you can provide a function to compute partial derivatives analytically. Typically, the problem is solved more accurately and efficiently if such a function is provided.

To solve the Eq. 1-2 using analytically determined gradients, do the following:

Step 1: Write an M-file for the objective function and gradient

```
function [f,G] = objfungrad(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
% Gradient of the objective function
t = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
G = [ t + exp(x(1)) * (8*x(1) + 4*x(2)),
    exp(x(1))*(4*x(1)+4*x(2)+2)];
```

Step 2: Write an M-file for the nonlinear constraints and the gradients of the nonlinear constraints

```
function [c,ceq,DC,DCeq] = confungrad(x)
c(1) = 1.5 + x(1) * x(2) - x(1) - x(2);   %inequality constraints
c(2) = -x(1) * x(2)-10;
% Gradient of the constraints
DC= [x(2)-1, -x(2);
      x(1)-1, -x(1)];
% No nonlinear equality constraints
ceq=[];
DCeq = [ ];
```

`G` contains the partial derivatives of the objective function, `f`, returned by `objfungrad(x)`, with respect to each of the elements in `x`:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) + e^{x_1}(8x_1 + 4x_2) \\ \\ e^{x_1}(4x_1 + 4x_2 + 2) \end{bmatrix} \tag{1-4}$$

The columns of DC contain the partial derivatives for each respective constraint (i.e., the ith column of DC is the partial derivative of the ith constraint with respect to x). So in the above example, DC is

$$\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_2}{\partial x_1} \\ \\ \frac{\partial c_1}{\partial x_2} & \frac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ \\ x_1 - 1 & -x_1 \end{bmatrix} \tag{1-5}$$

Since you are providing the gradient of the objective n objfungrad.m and the gradient of the constraints in confungrad.m, you *must* tell fmincon that these M-files contain this additional information. Use optimset to turn the parameters GradObj and GradConstr to 'on' in our already existing options structure

```
options = optimset(options,'GradObj','on','GradConstr','on');
```

If you do not set these parameters to 'on' in the options structure, fmincon will not use the analytic gradients.

The arguments lb and ub place lower and upper bounds on the independent variables in x. In this example we have no bound constraints and so they are both set to [].

Step 3: Invoke constrained optimization routine

```
x0 = [-1,1];            % Starting guess
options = optimset('LargeScale','off');
options = optimset(options,'GradObj','on','GradConstr','on');
lb = [ ]; ub = [ ];   % No upper or lower bounds
[x,fval] = fmincon('objfungrad',x0,[],[],[],[],lb,ub,...
   'confungrad',options)
[c,ceq] = confungrad(x) % Check the constraint values at x
```

After 20 function evaluations, the solution produced is

```
x =
    -9.5474    1.0474
fval =
     0.0236
c =
     1.0e-14 *
     0.1110
    -0.1776
ceq =
     []
```

Gradient Check: Analytic Versus Numeric

When analytically determined gradients are provided, you can compare the supplied gradients with a set calculated by finite-difference evaluation. This is particularly useful for detecting mistakes in either the objective function or the gradient function formulation.

If such gradient checks are desired, set the `DerivativeCheck` parameter to `'on'` using `optimset`:

```
options = optimset(options,'DerivativeCheck','on');
```

The first cycle of the optimization checks the analytically determined gradients (of the objective function and, if they exist, the nonlinear constraints). If they do not match the finite-differencing gradients within a given tolerance, a warning message indicates the discrepancy and gives the option to abort the optimization or to continue.

Equality Constrained Example

For routines that permit equality constraints, nonlinear equality constraints must be computed in the M-file with the nonlinear inequality constraints. For linear equalities, the coefficients of the equalities are passed in through the matrix `Aeq` and the right-hand-side vector `beq`.

For example, if you have the nonlinear equality constraint $x_1^2 + x_2 = 1$ and the nonlinear inequality constraint $x_1 x_2 \geq -10$, rewrite them as

$$x_1^2 + x_2 - 1 = 0$$
$$-x_1 x_2 - 10 \leq 0$$

and then, to solve the problem:

Step 1: Write an M-file objfun.m

```
function f = objfun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
```

Step 2: Write an M-file confuneq.m for the nonlinear constraints

```
function [c, ceq] = confuneq(x)
% nonlinear inequality constraints
c = -x(1)*x(2) - 10;
% nonlinear equality constraints
ceq = x(1)^2 + x(2) - 1;
```

Step 3: Invoke constrained optimization routine

```
x0 = [-1,1];            % Make a starting guess at the solution
options = optimset('LargeScale','off');
[x,fval] = fmincon('objfun',x0,[],[],[],[],[],[],...
   'confuneq',options)
[c,ceq] = confuneq(x) % Check the constraint values at x
```

After 21 function evaluations, the solution produced is

```
x =
   –0.7529    0.4332
fval =
    1.5093
c =
    –9.6739
ceq =
  4.0684e–010
```

Note that ceq is equal to 0 within the default tolerance on the constraints of 1.0e–006 and that c is less than or equal to zero as desired.

Maximization

The optimization functions fminbnd, fminsearch, fminunc, fmincon, fgoalattain, fminimax, lsqcurvefit, and lsqnonlin all perform minimization of the objective function, $f(x)$. Maximization is achieved by supplying the routines with $-f(x)$. Similarly, to achieve maximization for quadprog supply –H and –f, and for linprog supply –f.

Greater than Zero Constraints

The Optimization Toolbox assumes nonlinear inequality constraints are of the form $C_i(x) \leq 0$. Greater than zero constraints are expressed as less than zero constraints by multiplying them by –1. For example, a constraint of the form $C_i(x) \geq 0$ is equivalent to the constraint $-C_i(x) \leq 0$; a constraint of the form $C_i(x) \geq b$ is equivalent to the constraint $-C_i(x) + b \leq 0$.

Additional Arguments: Avoiding Global Variables

Parameters that would otherwise have to be declared as global can be passed directly to M-file functions using additional arguments at the end of the calling sequence.

For example, entering a number of variables at the end of the call to fsolve:

```
[x,fval] = fsolve('objfun',x0,options,P1,P2,...)
```

passes the arguments directly to the functions `objfun` when it is called from inside `fsolve`,

```
F = objfun(x,P1,P2, ... )
```

Consider, for example, finding zeros of the function `ellipj(u,m)`. The function needs parameter `m` as well as input `u`. To look for a zero near `u = 3`, for `m = 0.5`

```
m = 0.5;
options = optimset('Display','off'); % Turn off Display
x = fsolve('ellipj',3,options,m)
```

returns

```
x =
    3.7081
```

Then, solve for the function `ellipj`.

```
f = ellipj(x,m)
f =
    –2.9925e–008
```

The call to `optimset` to get the default options for `fsolve` implies that default tolerances are used and that analytic gradients are not provided.

Multiobjective Examples

The previous examples involved problems with a single objective function. This section demonstrates solving problems with multiobjective functions using `lsqnonlin`, `fminimax`, and `fgoalattain`. Included is an example of how to optimize parameters in a Simulink model.

Simulink Example

Let's say that you want to optimize the control parameters in the Simulink model `optsim.mdl`. (This model can be found in the Optimization Toolbox directory. Note that Simulink must be installed on your system to load this model.) The model includes a nonlinear process plant modeled as a Simulink block diagram shown in Figure 1-1.

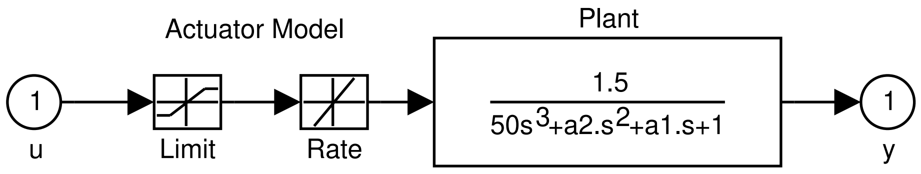


Figure 1-1: Plant with Actuator Saturation

The plant is an under-damped third-order model with actuator limits. The actuator limits are a saturation limit and a slew rate limit. The actuator saturation limit cuts off input values greater than 2 units or less than –2 units. The slew rate limit of the actuator is 0.8 units/sec. The open-loop response of the system to a step input is shown in Figure 1-2. (You can see this response by opening the model (type `optsim` at the command line), opening the Scope block, and running the simulation. The response plots to the Scope.)

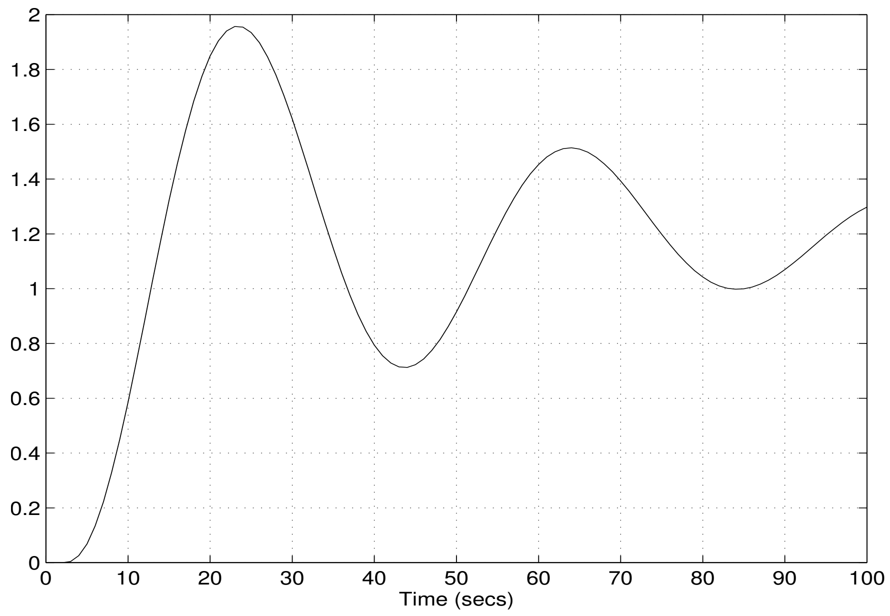


Figure 1-2: Open-Loop Response

The problem is to design a feedback control law that tracks a unit step input to the system. The closed-loop plant is entered in terms of the blocks where the plant and actuator have been placed in a hierarchical Subsystem block. A Scope block displays output trajectories during the design process. See Figure 1-3.

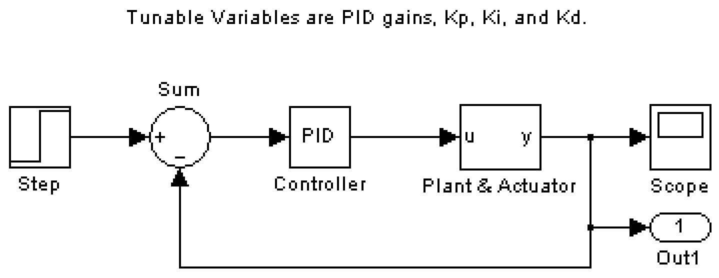


Figure 1-3: Closed-Loop Model

One way to solve this problem is to minimize the error between the output and the input signal. The variables are the parameters of the PID controller. If you only need to minimize the error at one time unit, it would be a single objective function. But the goal is to minimize the error for all time steps from 0 to 100, thus producing a multiobjective function (one function for each time step).

The routine `lsqnonlin` is used to perform a least squares fit on the tracking of the output. This is defined via a MATLAB function in the file `tracklsq.m` shown below that defines the error signal. The error signal is `yout`, the output computed by calling `sim`, minus the input signal 1.

The function `tracklsq` must run the simulation. The simulation can be run either in the base workspace or the current workspace, i.e., the workspace of the function calling `sim`, which in this case is `tracklsq`'s workspace. In this example, the `simset` command is used to tell `sim` to run the simulation in the current workspace by setting `'SrcWorkspace'` to `'Current'`.

To run the simulation in `optsim`, the variables `Kp`, `Ki`, `Kd`, `a1`, and `a2` (`a1` and `a2` are variables in the Plant block) must all be defined. `Kp`, `Ki`, and `Kd` are the variables we are optimizing. You can initialize `a1` and `a2` before calling `lsqnonlin` and then pass these two variables as additional arguments. `lsqnonlin` will then pass `a1` and `a2` to `tracklsq` each time it is called so you do not have to use global variables.

After choosing a solver using the `simset` function, the simulation is run using `sim`. The simulation is performed using a fixed-step fifth-order method to 100 seconds. When the simulation completes, the variables `tout`, `xout`, and `yout` are now in the current workspace (that is, the workspace of `tracklsq`). The

Outport block is used in the block diagram model to put yout into the current workspace at the end of the simulation.

Step 1: Write an M-file tracklsq.m

```
function F = tracklsq(pid,a1,a2)
Kp = pid(1);   % Move variables into model parameter names
Ki = pid(2);
Kd = pid(3);
% Choose solver and set model workspace to this function
opt = simset('solver','ode5','SrcWorkspace','Current');
[tout,xout,yout] = sim('optsim',[0 100],opt);
F = yout-1;    % Compute error signal
```

Step 2: Invoke optimization routine

```
optsim % load the model
pid0 = [0.63 0.0504 1.9688] % Set initial values
a1 = 3; a2 = 43; % Initialize Plant variables in model
options = optimset('LargeScale','off','Display','iter',...
      'TolX',0.001,'TolFun',0.001);
pid = lsqnonlin('tracklsq', pid0, [], [], options, a1, a2)
% put variables back in the base workspace
Kp = pid(1); Ki = pid(2); Kd = pid(3);
```

The variable options passed to lsqnonlin defines the criteria and display characteristics. In this case you ask for output, use the medium-scale algorithm, and give termination tolerances for the step and objective function on the order of 0.001.

The optimization gives the solution for the Proportional, Integral, and Derivative (`Kp`, `Ki`, `Kd`) gains of the controller after 73 function evaluations:

```
                                                    Directional
  Iteration  Func-count  Residual   Step-size derivative    Lambda
      1          3        8.66531       1        –3.48
      2         10        6.78831       1       –0.0634        3.4355
      3         19        5.99204     5.5       –0.0446        0.28612
      4         28        4.74992     5.78      –0.0213      0.0227966
      5         36        4.51795     1.25       0.0222      0.0744258
      6         43        4.5115      0.58     –0.00633        0.03445
      7         51        4.49455     2.99     0.000688       0.017225
      8         58        4.4836     0.915      0.00203      0.0180998
      9         66        4.47724     1.22     0.000845     0.00904992
     10         73        4.47405    0.801     –0.00072      0.0113409
  Optimization terminated successfully:
   Gradient in the search direction less than tolFun
   Gradient less than 10*(tolFun+tolX)
  pid =
      2.9108    0.1443   12.8161
```

The resulting closed-loop step response is shown in Figure 1-4.

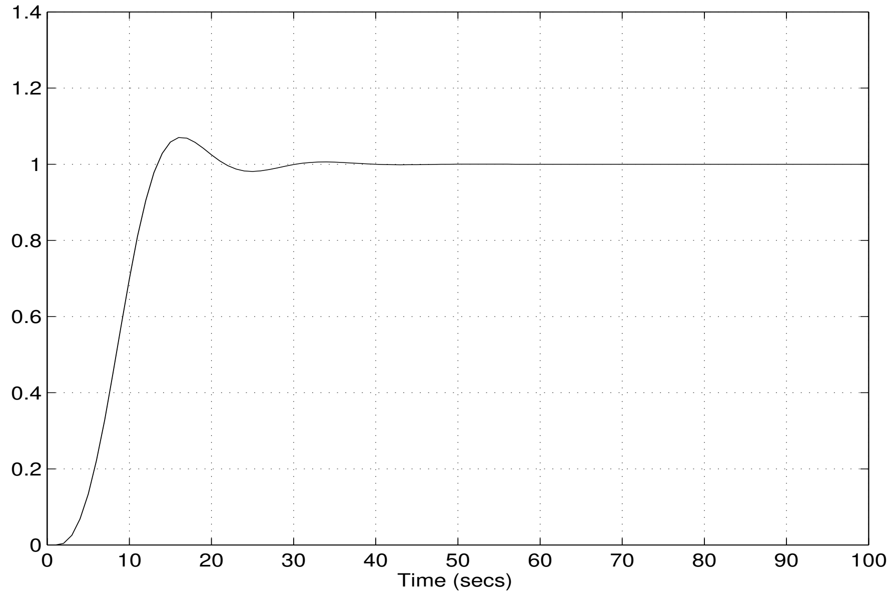


Figure 1-4: Closed-Loop Response using lsqnonlin

Note: The call to `sim` results in a call to one of the Simulink ordinary differential equation (ODE) solvers. A choice must be made about the type of solver to use. From the optimization point of view, a fixed-step solver is the best choice if that is sufficient to solve the ODE. However, in the case of a stiff system, a variable-step method may be required to solve the ODE. The numerical solution produced by a variable-step solver, however, is not a smooth function of parameters because of step-size control mechanisms. This lack of smoothness may prevent the optimization routine from converging. The lack of smoothness is not introduced when a fixed-step solver is used. (For a further explanation, see *Solving Ordinary Differential Equations I -- Nonstiff Problems*, by E. Hairer, S.P. Norsett, G. Wanner, Springer-Verlag, pages 183-184.) The *NCD Blockset* is recommended for solving multiobjective optimization problems in conjunction with variable-step solvers in Simulink; it provides a special numeric gradient computation that works with Simulink and avoids introducing the lack of smoothness problem.

Another solution approach is to use the fminimax function. In this case, rather than minimizing the error between the output and the input signal, you minimize the maximum value of the output at any time t between 0 and 100. Then in the function trackmmobj the objective function is simply the output yout returned by the sim command. But minimizing the maximum output at all time steps may force the output far below unity for some time steps. To keep the output above 0.95 after the first 20 seconds, in the constraint function trackkmmcon add a constraint yout >= 0.95 from t=20 to t=100. Because constraints must be in the form g<=0, the constraint in the function is g = –yout(20:100)+.95.

Both trackmmobj and trackmmcon use the result yout from sim, calculated from the current pid values. The nonlinear constraint function is *always* called immediately after the objective function in fmincon, fminimax, fgoalattain, and fseminf with the same values. Thus you can avoid calling the simulation twice by using assignin to assign the current value of F to the variable F_TRACKMMOBJ in the base workspace. Then the first step in trackmmcon is to use evalin to evaluate the variable F_TRACKMMOBJ in the base workspace, and assign the result to F locally in trackmmcon.

Step 1: Write an M-file trackmmobj.m to compute objective function

```
function F = trackmmobj(pid,a1,a2)
Kp = pid(1);
Ki = pid(2);
Kd = pid(3);
% Compute function value
opt = simset('solver','ode5','SrcWorkspace','Current');
[tout,xout,yout] = sim('optsim',[0 100],opt);
F = yout;
assignin('base','F_TRACKMMOBJ',F);
```

Step 2: Write an M-file trackmmcon.m to compute nonlinear constraints

```
function [c,ceq] = trackmmcon(pid,a1,a2)
F = evalin('base','F_TRACKMMOBJ');
% Compute constraints
c = –F(20:100)+.95;
ceq = [ ];
```

Note that `fminimax` will pass `a1` and `a2` to the objective and constraint values, so `trackmmcon` needs input arguments for these variables even though it does not use them.

Step 3: Invoke constrained optimization routine

```
optsim
pid0 = [0.63 0.0504 1.9688]
a1 = 3; a2 = 43;
options = optimset('Display','iter',...
             'TolX',0.001,'TolFun',0.001);
pid = fminimax('trackmmobj',pid0,[],[],[],[],[],[],...
          'trackmmcon',options,a1,a2)
% put variables back in the base workspace
Kp = pid(1); Ki = pid(2); Kd = pid(3);
```

resulting in

```
                         Max                      Directional
 Iter   F-count  {F,constraints}  Step-size  derivative    Procedure
   1       5           1.12            1           1.18
   2      11          1.264            1         -0.172
   3      17          1.055            1        -0.0128   Hessian
                                                          modified
                                                           twice
   4      23          1.004            1       3.49e-005  Hessian
                                                          modified
   5      29         0.9997            1      -1.36e-006  Hessian
                                                          modified
                                                           twice
Optimization terminated successfully:
 Search direction less than 2*options.TolX and
  maximum constraint violation is less than options.TolCon
Active Constraints:
     1
    14
   182
pid =
    0.5894    0.0605    5.5295
```

The last value shown in the `MAX{F,constraints}` column of the output shows the maximum value for all the time steps is 0.9997. The closed loop response with this result is shown in Figure 1-5.

This solution differs from the `lsqnonlin` solution as you are solving different problem formulations.

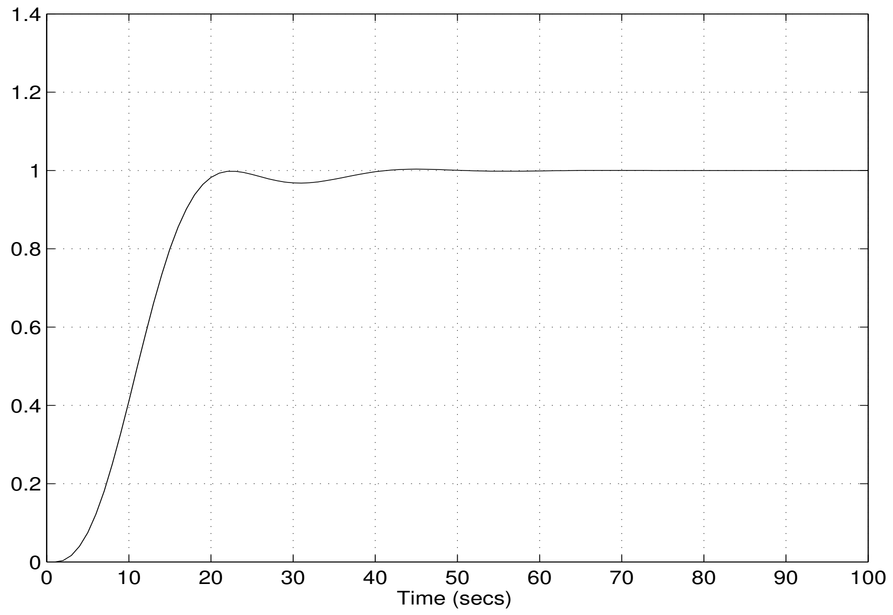


Figure 1-5: Closed-Loop Response using fminimax

Signal Processing Example

Consider designing a linear-phase Finite Impulse Response (FIR) filter. The problem is to design a lowpass filter with magnitude one at all frequencies between 0 and 0.1 Hz and magnitude zero between 0.15 and 0.5 Hz.

The frequency response *H(f)* for such a filter is defined by

$$
\begin{aligned}
H(f) &= \sum_{n=0} h(n)e^{-j2\pi fn} \\
&= A(f)e^{-j2\pi fM} \\
A(f) &= \sum_{n=0}^{M-1} a(n)\cos(2\pi fn)
\end{aligned}
\tag{1-6}
$$

where *A(f)* is the magnitude of the frequency response. One solution is to apply a goal attainment method to the magnitude of the frequency response. Given a function that computes the magnitude, the function `fgoalattain` will attempt to vary the magnitude coefficients *a(n)* until the magnitude response matches the desired response within some tolerance. The function that computes the magnitude response is given in `filtmin.m`. This function takes `a`, the magnitude function coefficients, and `w`, the discretization of the frequency domain we are interested in.

To set up a goal attainment problem, you must specify the `goal` and `weights` for the problem. For frequencies between 0 and 0.1, the goal is one. For frequencies between 0.15 and 0.5, the goal is zero. Frequencies between 0.1 and 0.15 are not specified so no goals or weights are needed in this range.

This information is stored in the variable `goal` passed to `fgoalattain`. The length of `goal` is the same as the length returned by the function `filtmin`. So that the goals are equally satisfied, usually `weight` would be set to `abs(goal)`. However, since some of the goals are zero, the effect of using `weight=abs(goal)` will force the objectives with `weight` 0 to be satisfied as hard constraints, and the objectives with `weight` 1 possibly to be underattained (see "The Goal Attainment Method" section of the *Introduction to Algorithms* chapter). Because all the goals are close in magnitude, using a `weight` of unity for all goals will give them equal priority. (Using `abs(goal)` for the weights is more important when the magnitude of `goal` differs more significantly.) Also, setting

```
options = optimset('GoalsExactAchieve',length(goal));
```

specifies that each objective should be as near as possible to its goal value (neither greater nor less than).

Step 1: Write an M-file filtmin.m

```
function y = filtmin(a,w)
n = length(a);
y = cos(w'*(0:n–1)*2*pi)*a ;
```

Step 2: Invoke optimization routine

```
% Plot with initial coefficients
a0 = ones(15,1);
incr = 50;
w = linspace(0,0.5,incr);

y0 = filtmin(a0,w);
clf, plot(w,y0.'–:');
drawnow;

% Set up the goal attainment problem
w1 = linspace(0,0.1,incr) ;
w2 = linspace(0.15,0.5,incr);
w0 = [w1 w2];
goal = [1.0*ones(1,length(w1)) zeros(1,length(w2))];
weight = ones(size(goal));

% Call fgoalattain
options = optimset('GoalsExactAchieve',length(goal));
[a,fval,attainfactor,exitflag] = fgoalattain('filtmin',...
    a0,goal,weight,[],[],[],[],[],[],[],options,w0);

% Plot with the optimized (final) coefficients
y = filtmin(a,w);
hold on, plot(w,y,'r')
axis([0 0.5 –3 3])
xlabel('Frequency (Hz)')
ylabel('Magnitude Response (dB)')
legend('initial', 'final')
grid on
```

Compare the magnitude response computed with the initial coefficients and the final coefficients (Figure 1-6). Note that you could use the `remez` function in the *Signal Processing Toolbox* to design this filter.

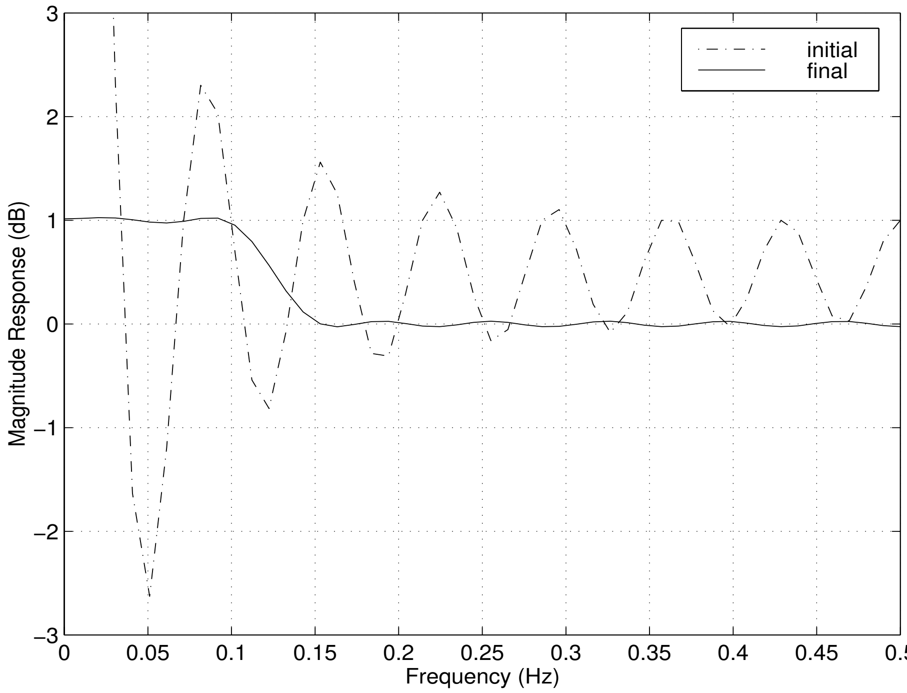


Figure 1-6: Magnitude Response with Initial and Final Magnitude Coefficients

Large-Scale Examples

Some of the optimization functions include algorithms for continuous optimization problems especially targeted to large problems with sparsity or structure. The main large-scale algorithms are iterative, i.e., a sequence of approximate solutions is generated. In each iteration a linear system is (approximately) solved. The linear systems are solved using the sparse matrix capabilities of MATLAB and a variety of sparse linear solution techniques, both iterative and direct.

Generally speaking the large-scale optimization methods preserve structure and sparsity, using exact derivative information wherever possible. To solve the large-scale problems efficiently, some problem formulations are restricted (such as only solving overdetermined linear or nonlinear systems), or require additional information (e.g., the nonlinear minimization algorithm requires the gradient be computed in the user-supplied function).

Not all possible problem formulations are covered by the large-scale algorithms. The following table describes what functionality is covered by the large-scale methods. For example, for `fmincon`, the large-scale algorithm covers the case where there are only bound constraints or only linear equalities. For each problem formulation, the third column describes what additional information is needed for the large-scale algorithms. For `fminunc` and `fmincon`, the gradient must be computed along with the objective in the user-supplied function (the gradient is not required for the medium-scale algorithms).

Since these methods can also be used on small- to medium-scale problems that are not necessarily sparse, the last column of the table emphasizes what conditions are needed for large-scale problems to run efficiently without exceeding your computer system's memory capabilities, e.g., the linear constraint matrices should be sparse. For smaller problems the conditions in the last column are unnecessary.

Several examples follow this table to further clarify the contents of the table.

Table 1-4: Large-Scale Problem Coverage and Requirements

Function	Problem Formulations	Additional Information Needed	For Large Problems
`fminunc`	$\min_x f(x)$	Must provide gradient for `f(x)` in `fun`.	• Provide sparsity structure of the Hessian, or compute the Hessian in `fun`. • The Hessian should be sparse.
`fmincon`	• $\min_x f(x)$ such that $l \le x \le u$ where $l < u$ • $\min_x f(x)$ such that $Aeq \cdot x = beq$ *Aeq* is an *m*-by-*n* matrix where $m \le n$.	Must provide gradient for `f(x)` in `fun`.	• Provide sparsity structure of the Hessian, or compute the Hessian in `fun`. • The Hessian should be sparse.
`lsqnonlin`	• $\min_x \frac{1}{2}\Vert F(x)\Vert_2^2 = \frac{1}{2}\sum_i F_i(x)^2$ • $\min_x \frac{1}{2}\Vert F(x)\Vert_2^2 = \frac{1}{2}\sum_i F_i(x)^2$ such that $l \le x \le u$ where $l < u$ *F(x)* must be overdetermined (have at least as many equations as variables).	Not applicable.	• Provide sparsity structure of the Jacobian, or compute the Jacobian in `fun`. • The Jacobian should be sparse.

Table 1-4: Large-Scale Problem Coverage and Requirements (Continued)

Function	Problem Formulations	Additional Information Needed	For Large Problems
`lsqcurvefit`	• $\min_{x} \frac{1}{2}\Vert F(x, xdata) - ydata\Vert_2^2$ • $\min_{x} \frac{1}{2}\Vert F(x, xdata) - ydata\Vert_2^2$ such that $l \le x \le u$ where $l < u$ *F(x,xdata)* must be overdetermined (have at least as many equations as variables).	Not applicable.	• Provide sparsity structure of the Jacobian, or compute the Jacobian in `fun`. • The Jacobian should be sparse.
`fsolve`	$F(x) = 0$ *F(x)* must be overdetermined (have at least as many equations as variables).	Not applicable.	• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.
`lsqlin`	$\min_{x} \Vert C \cdot x - d\Vert_2^2$ such that $l \le x \le u$ where $l < u$ C is an m-by-n matrix where $m \ge n$, i.e., the problem must be overdetermined.	Not applicable.	C should be sparse.

Table 1-4: Large-Scale Problem Coverage and Requirements (Continued)

Function	Problem Formulations	Additional Information Needed	For Large Problems
`linprog`	$\min_x f^T x$ such that $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$	Not applicable.	A and Aeq should be sparse.
`quadprog`	• $\min_x \frac{1}{2}x^T Hx + f^T x$ such that $l \le x \le u$ where $l < u$ • $\min_x \frac{1}{2}x^T Hx + f^T x$ such that $Aeq \cdot x = beq$ Aeq is an m-by-n matrix where $m \le n$.	Not applicable.	• H should be sparse. • Aeq should be sparse.

In the examples below, many of the M-file functions are available in the Optimization Toolbox `optim` directory. Most of these do not have a fixed problem size, i.e., the size of your starting point `xstart` will determine the size problem that is computed. If your computer system cannot handle the size suggested in the examples below, use a smaller dimension start point to run the problems. If the problems have upper or lower bounds or equalities, you will have to adjust the size of those vectors or matrices as well.

Nonlinear Equations with Jacobian

Consider the problem of finding a solution to a system of nonlinear equations whose Jacobian is sparse. The dimension of the problem in this example is 1000. The goal is to find x such that $F(x) = 0$. Assuming $n=1000$, the nonlinear equations are

$$F(1) = 3x_1 - 2x_1^2 - 2x_2 + 1$$

$$F(i) = 3x_i - 2x_i^2 - x_{i-1} - 2x_{i+1} + 1$$

$$F(n) = 3x_n - 2x_n^2 - x_{n-1} + 1$$

To solve a large nonlinear system of equations, $F(x) = 0$, use the large-scale method available in `fsolve`.

Step 1: Write an M-file nlsf1.m that computes the objective function values and the Jacobian

```
function [F,J] = nlsf1(x);
% Evaluate the vector function
n = length(x);
F = zeros(n,1);
i = 2:(n–1);
F(i) = (3–2*x(i)).*x(i)–x(i–1)–2*x(i+1)1+ 1;
F(n) = (3–2*x(n)).*x(n)–x(n–1) + 1;
F(1) = (3–2*x(1)).*x(1)–2*x(2) + 1;
% Evaluate the Jacobian if nargout > 1
if nargout > 1
   d = –4*x + 3*ones(n,1); D = sparse(1:n,1:n,d,n,n);
   c = –2*ones(n–1,1); C = sparse(1:n–1,2:n,c,n,n);
   e = –ones(n-1,1); E = sparse(2:n,1:n-1,e,n,n);
   J = C + D + E;
end
```

Step 2: Call the system of equations solve routine

```
xstart = –ones(1000,1);
fun = 'nlsf1';
options = optimset('Display','iter','Jacobian','on');
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

A starting point is given as well as the function name. The large-scale method is the default, so it is not necessary to specify this in the `options` argument. Output at each iteration is specified. Finally, so that `fsolve` will use the Jacobian information available in `nlsf1.m`, you need to turn the `Jacobian` parameter `'on'` using `optimset`.

These commands display this output:

```
                                                Norm of  First-order
Iteration  Func-count        f(x)                  step   optimality  CG-iterations
        1           2        1011                     1           19              0
        2           3     16.1942               7.91898         2.35              3
        3           4   0.0228027               1.33142        0.291              3
        4           5 0.000103359             0.0433329       0.0201              4
        5           6 7.3792e-007             0.0022606     0.000946              4
        6           7 4.02299e-010          0.000268381    4.12e-005              5

   Optimization terminated successfully:Relative function value
   changing by less than OPTIONS.TolFun
```

A linear system is (approximately) solved in each major iteration using the preconditioned conjugate gradient method. The default value for the `PrecondBandWidth` is 0 in options, so a diagonal preconditioner is used. (The `PrecondBandWidth` specifies the bandwidth of the preconditioning matrix. A bandwidth of 0 means there is only one diagonal in the matrix.)

From the first-order optimality values, fast linear convergence occurs. The number of CG iterations required per major iteration is low, at most 5 for a problem of 1000 dimensions, implying the linear systems are not very difficult to solve in this case (though more work is required as convergence progresses).

It is possible to override the default choice of preconditioner (diagonal) by choosing a banded preconditioner through the use of the parameter `PrecondBandWidth`. If you want to use a tridiagonal preconditioner, i.e., a preconditioning matrix with three diagonals (or bandwidth of one), set `PrecondBandWidth` to the value `1`:

```
options = optimset('Display','iter','Jacobian','on',...
                   'PrecondBandWidth',1) ;
[x,fval,exitflag,output] = fsolve(fun,xstart,options) ;
```

In this case the output is

```
Iteration  Func-count         f(x)     Norm of First-order CG-iterations
                                          step  Optimality
        1           2         1011           1          19            0
        2           3      16.0839     7.92496        1.92            1
        3           4    0.0458181      1.3279       0.579            1
        4           5  0.000101184   0.0631898      0.0203            2
        5           6 3.16615e-007  0.00273698     0.00079            2
        6           7 9.72481e-010  0.00018111   5.82e-005            2

   Optimization terminated successfully:
    Relative function value changing by less than OPTIONS.TolFun
```

Note that although the same number of iterations takes place, the number of PCG iterations has dropped, so less work is being done per iteration.

Nonlinear Equations with Jacobian Sparsity Pattern

In the preceding example the function `nlsf1` computes the Jacobian `J`, a sparse matrix, along with the evaluation of `F`. What if the code to compute the Jacobian is not available? By default, if you do not indicate the Jacobian can be computed in `nlsf1` (using the `Jacobian` parameter in `options`), `fsolve`, `lsqnolin`, and `lsqcurvefit` will instead use finite-differencing to approximate the Jacobian.

In order for this finite-differencing to be as efficient as possible, the sparsity pattern of the Jacobian should be supplied, using the `JacobPattern` parameter in `options`. That is, supply a sparse matrix `Jstr` whose nonzero entries correspond to nonzeroes of the Jacobian for all x. Indeed, the nonzeroes of `Jstr` can correspond to a superset of the nonzero locations of J; however, in general the computational cost of the sparse finite-difference procedure will increase with the number of nonzeroes of `Jstr`.

Providing the sparsity pattern can drastically reduce the time needed to compute the finite-differencing on large problems. If the sparsity pattern is not provided (and the Jacobian is not computed in objective function either) then, in this problem `nlsfs1`, the finite-differencing code will attempt to compute all 1000-by-1000 entries in the Jacobian. But in this case there are only 2998 nonzeros, substantially less than the 1,000,000 possible nonzeros the finite-differencing code will attempt to compute. In other words, this problem is solvable if the sparsity pattern is provided. If not, most computers will run out of memory when the full dense finite-differencing is attempted. On most small problems, it is not essential to provide the sparsity structure.

Suppose the sparse matrix `Jstr`, computed previously, has been saved in file `nlsdat1.mat`. The following driver calls `fsolve` applied to `nlsf1a` which is the same as `nlsf1` except only the function values are returned; sparse finite-differencing is used to estimate the sparse Jacobian matrix as needed.

Step 1: Write an M-file nlsf1a.m that computes the objective function values

```
function F = nlsf1a(x);
% Evaluate the vector function
n = length(x);
F = zeros(n,1);
i = 2:(n–1);
F(i) = (3–2*x(i)).*x(i)–x(i–1)–2*x(i+1) + 1;
F(n) = (3–2*x(n)).*x(n)–x(n–1) + 1;
F(1) = (3–2*x(1)).*x(1)–2*x(2) + 1;
```

Step 2: Call the system of equations solve routine

```
xstart = –ones(1000,1);
fun = 'nlsf1a';
load nlsdat1   % Get Jstr
options = optimset('Display','iter','JacobPattern',Jstr,...
                   'PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case, the output displayed is

```
                                            Norm of  First-order
Iteration  Func-count          f(x)            step   optimality CG-iterations
        1           6          1011               1           19             0
        2          11       16.0839         7.92496         1.92             1
        3          16     0.0458181          1.3279        0.579             1
        4          21   0.000101184       0.0631897       0.0203             2
        5          26  3.16615e–007      0.00273698      0.00079             2
        6          31  9.72482e–010      0.00018111    5.82e–005             2

  Optimization terminated successfully:
   Relative function value changing by less than OPTIONS.TolFun
```

Alternatively, it is possible to choose a sparse direct linear solver (i.e., a sparse QR factorization) by indicating a “complete” preconditioner, i.e., if we set

PrecondBandWidth to Inf, then a sparse direct linear solver will be used instead of a preconditioned conjugate gradient iteration

```
xstart = –ones(1000,1);
fun = 'nlsf1a';
load nlsdat1   % Get Jstr
options = optimset('Display','iter','JacobPattern',Jstr,...
                   'PrecondBandWidth',inf);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

and the resulting display is

```
                                         Norm of  First-order  CG-iterations
Iteration  Func-count           f(x)        step   optimality
        1           6           1011           1           19              0
        2          11        15.9018     7.92421         1.89              1
        3          16      0.0128163     1.32542       0.0746              1
        4          21  1.73537e–008    0.0397925     0.000196              1
        5          26  1.13136e–018 4.55542e–005    2.76e–009              1

   Optimization terminated successfully:
    Relative function value changing by less than OPTIONS.TolFun
```

When the sparse direct solvers are used, the CG iteration will be 1 for that (major) iteration, as shown in the output under CG-iterations. Notice that the final optimality and *f(x)* value (which for fsolve, *f(x)*, is the sum-of-the-squares of the function values) are closer to zero than using the PCG method, which is often the case.

Nonlinear Least-Squares with Full Jacobian Sparsity Pattern

The large-scale methods in lsqnonlin, lsqcurvefit, and fsolve can be used with small- to medium-scale problems without computing the Jacobian in fun or providing the Jacobian sparsity pattern. (This example also applies to the case of using fmincon or fminunc without computing the Hessian or supplying the Hessian sparsity pattern.) How small is small- to medium-scale? No absolute answer is available, as it depends on the amount of virtual memory available in your computer system configuration.

Suppose your problem has m equations and n unknowns. If the command J = sparse(ones(m,n)) causes an OUT OF MEMORY error on your machine, then this is certainly too large a problem. If it does not result in an error, the problem may still be too large, but you can only find out by running it and seeing if MATLAB is able to run within the amount of virtual memory available on your system.

Let's say you have a small problem with 10 equations and 2 unknowns, such as find x that minimizes

$$\sum_{k=1}^{10} (2+2k-e^{kx_1}-e^{kx_2})^2$$

starting at the point `x = [0.3, 0.4]`.

Because `lsqnonlin` assumes that the sum-of-squares is not explicitly formed in the user function, the function passed to `lsqnonlin` should instead compute the vector valued function $F_k(x) = 2+2k-e^{kx_1}-e^{kx_2}$

for $k = 1$ to 10 (that is, `F` should have `k` components).

Step 1: Write an M-file myfun.m that computes the objective function values

```
function F = myfun(x)
k = 1:10;
F = 2 + 2*k–exp(k*x(1))–exp(k*x(2));
```

Step 2: Call the nonlinear least-squares routine

```
x0 = [0.3 0.4]            % Starting guess
[x,resnorm] = lsqnonlin('myfun',x0)     % Invoke optimizer
```

Since the Jacobian is not computed in `myfun.m` (and the `Jacobian` parameter in `options` is `'off'` by default), and no Jacobian sparsity pattern is provided using the `JacobPattern` parameter in `options`, `lsqnonlin` will call the large-scale method, the default for `lsqnonlin`, with `JacobPattern` set to `Jstr = sparse(ones(10,2))`. When the finite-differencing routine is called the first time, it will detect that `Jstr` is actually a dense matrix, i.e., that no speed benefit is derived from storing it as a sparse matrix, and from then on will use `Jstr = ones(10,2)` (a full matrix) for the optimization computations.

After about 24 function evaluations, this example gives the solution:

```
x =
     0.2578   0.2578
resnorm     % residual or sum of squares
resnorm =
     124.3622
```

Most computer systems will be able to handle much larger full problems, say into the 100's of equations and variables. But *if* there is some sparsity structure in the Jacobian (or Hessian) that can be taken advantage of, the large-scale methods will always run faster if this information is provided.

Nonlinear Minimization with Gradient and Hessian

This example involves a solving a nonlinear minimization problem with a tridiagonal Hessian matrix $H(x)$ first computed explicitly, and then by providing the Hessian's sparsity structure for the finite-differencing routine.

The problem is to find x to minimize

$$f(x) = \sum_{i=1}^{n-1} \left[(x_i^2)^{(x_{i+1}^2+1)} + (x_{i+1}^2)^{(x_i^2+1)} \right] \tag{1-7}$$

where $n=1000$.

Step 1: Write an M-file brownfgh.m that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix

This file is rather lengthy and is not included here. You can view the code with the command

```
type brownfgh
```

Because `brownfgh` computes the gradient and Hessian values as well as the objective function, you need to use `optimset` to indicate this information is available in `brownfgh` using the `GradObj` and `Hessian` parameters.

Step 2: Call a nonlinear minimization routine with a starting point xstart

```
n = 1000;
xstart = –ones(n,1);
xstart(2:2:n,1) = 1;
options = optimset('GradObj',...
                   'on','Hessian','on');
[x,fval,exitflag,output] = fminunc('brownfgh',xstart,options);
```

This 1000 variable problem is solved in 8 iterations and 7 conjugate gradient iterations with a positive `exitflag` indicating convergence; the final function value and measure of optimality at the solution `x` are both close to zero (for `fminunc`, the first order optimality is the gradient of the function, which is zero at a local minimum):

```
exitflag =
     1
fval =
  2.8709e–017
output.iterations
ans =
     8
output.cgiterations
ans =
     7
output.firstorderopt
ans =
  4.7948e–010
```

Nonlinear Minimization with Gradient and Hessian Sparsity Pattern

Next we solve the same problem but the Hessian matrix is now approximated by sparse finite-differences instead of explicit computation. To use the large-scale method in `fminunc`, you *must* compute the gradient in `fun`; it is *not optional* as in the medium-scale method.

The M-file function `brownfg` computes the objective function and gradient.

Step 1: Write an M-file brownfg.m that computes the objective function and the gradient of the objective

```
function [f,g] = brownfg(x,dummy)
% BROWNFG Nonlinear minimization test problem
%
% Evaluate the function.
n=length(x); y=zeros(n,1);
i=1:(n–1);
y(i)=(x(i).^2).^(x(i+1).^2+1) + ...
        (x(i+1).^2).^(x(i).^2+1);
  f=sum(y);
% Evaluate the gradient if nargout > 1
  if nargout > 1
     i=1:(n–1); g = zeros(n,1);
     g(i) = 2*(x(i+1).^2+1).*x(i).* ...
              ((x(i).^2).^(x(i+1).^2))+ ...
              2*x(i).*((x(i+1).^2).^(x(i).^2+1)).* ...
              log(x(i+1).^2);
     g(i+1) = g(i+1) + ...
              2*x(i+1).*((x(i).^2).^(x(i+1).^2+1)).* ...
              log(x(i).^2) + ...
              2*(x(i).^2+1).*x(i+1).* ...
              ((x(i+1).^2).^(x(i).^2));
  end
```

To allow efficient computation of the sparse finite-difference approximation of the Hessian matrix $H(x)$, the sparsity structure of H must be predetermined. In this case assume this structure, `Hstr`, a sparse matrix, is available in file `brownhstr.mat`. Using the `spy` command you can see that `Hstr` is indeed sparse (only 2998 nonzeros). Use `optimset` to set the `HessPattern` parameter to `Hstr`. When a problem as large as this has obvious sparsity structure, not setting the `HessPattern` parameter will require a huge amount of unnecessary memory and computation since `fminunc` will attempt to use finite-differencing on a full Hessian matrix of one million nonzero entries.

You must also set the `GradObj` parameter to `'on'` using `optimset` since the gradient is computed in `brownfg.m`. Then to execute `fminunc`:

Step 2: Call a nonlinear minimization routine with a starting point xstart

```
fun = 'brownfg';
load brownhstr   % get Hstr, structure of the Hessian
spy(Hstr) % view the sparsity structure of Hstr
n = 1000;
xstart = –ones(n,1);
xstart(2:2:n,1) = 1;
options = optimset('GradObj','on','HessPattern',Hstr);
[x,fval,exitflag,output] = fminunc(fun,xstart,options);
```

This 1000 variable problem is solved in 8 iterations and 7 conjugate gradient iterations with a positive `exitflag` indicating convergence; the final function value and measure of optimality at the solution `x` are both close to zero (for `fminunc`, the first order optimality is the gradient of the function, which is zero at a local minimum):

```
exitflag =
     1
fval =
  7.4738e–017
output.iterations
ans =
     8
output.cgiterations
ans =
     7
output.firstorderopt
ans =
  7.9822e–010
```

Nonlinear Minimization with Bound Constraints and Banded Preconditioner

The goal in this problem is to minimize the nonlinear function

$$f(x) = 1 + \sum_{i=1}^{n} \left|(3-2x_i)x_i - x_{i-1} - x_{i+1} + 1\right|^p + \sum_{i=1}^{\frac{n}{2}} \left|x_i + x_{i+n/2}\right|^p$$

such that $-10.0 \le x_i \le 10.0$, where n is 800 (n should be a multiple of 4), $p=7/3$, and $x_0 = x_{n+1} = 0$.

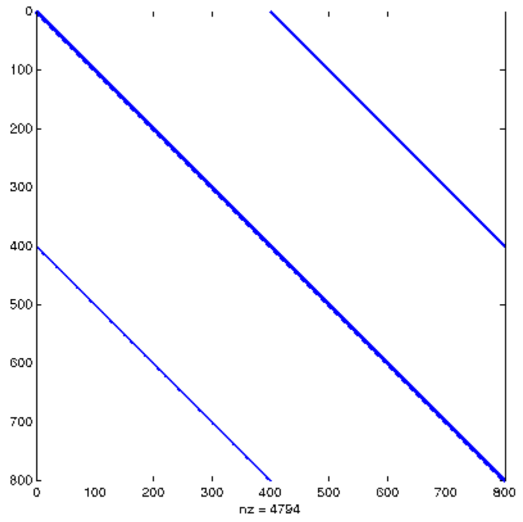
Step 1: Write an M-file tbroyfg.m that computes the objective function and the gradient of the objective

The M-file function, `tbroyfg.m`, computes the function value and gradient. This file is rather lengthy and is not included here. You can see the code for this function using the command

```
type tbroyfg
```

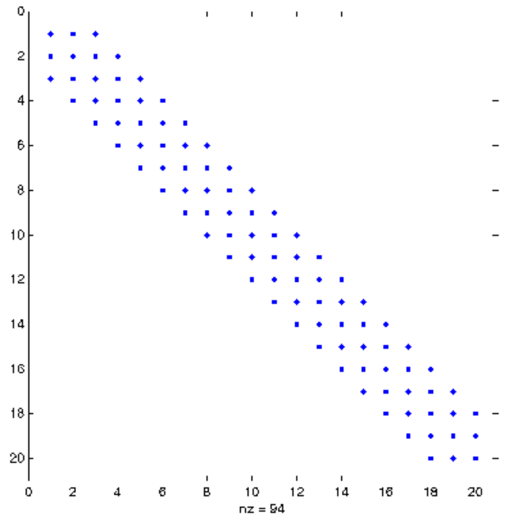
The sparsity pattern of the Hessian matrix has been predetermined and stored in the file `tbroyhstr.mat`. The sparsity structure for the Hessian of this problem is banded, as you can see in the spy plot below:

```
load tbroyhstr
spy(Hstr):
```



The center stripe is itself a 5-banded matrix

```
spy(Hstr(1:20,1:20))
```



Use `optimset` to set the `HessPattern` parameter to `Hstr`. When a problem as large as this has obvious sparsity structure, not setting the `HessPattern` parameter will require a huge amount of unnecessary memory and computation since `fmincon` will attempt to use finite-differencing on a full Hessian matrix of 640,000 nonzero entries.

You must also set the `GradObj` parameter to `'on'` using `optimset` since the gradient is computed in `tbroyfg.m`. Then to execute `fmincon`:

Step 2: Call a nonlinear minimization routine with a starting point xstart

```
fun = 'tbroyfg';
load tbroyhstr   % get Hstr, structure of the Hessian
n = 800;
xstart = –ones(n,1); xstart(2:2:n) = 1;
lb = –10*ones(n,1); ub = –lb;
options = optimset('GradObj','on','HessPattern',Hstr);
[x,fval,exitflag,output] = ...
   fmincon('tbroyfg',xstart,[],[],[],[],lb,ub,[],options);
```

After eight iterations, the `exitflag`, `fval` and `output` values are:

```
exitflag =
     1
fval =
  270.4790
output =
       iterations: 8
        funcCount: 8
     cgiterations: 18
    firstorderopt: 0.0163
        algorithm: 'large-scale: trust-region reflective Newton'
```

For bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" in Chapter 3, and `g` is the gradient.

Because of the 5-banded center stripe, you can improve the solution by using a 5-banded preconditioner instead of the default diagonal preconditioner. Using the `optimset` function, reset the `PrecondBandWidth` parameter to two and solve the problem again. (The bandwidth is the number of upper (or lower) diagonals, not counting the main diagonal.)

```
fun = 'tbroyfg';
load tbroyhstr % get Hstr, structure of the Hessian
n = 800;
xstart = –ones(n,1); xstart(2:2:n,1) = 1;
lb = –10*ones(n,1); ub = –lb;
options = optimset('GradObj','on','HessPattern',Hstr, ...
                   'PrecondBandWidth',2);
[x,fval,exitflag,output] = ...
   fmincon('tbroyfg',xstart,[],[],[],[],lb,ub,[],options);
```

The number of iterations actually goes up by two; however the total number of CG iterations drops from 18 to 15. The first-order optimality measure is reduced by a factor of `1e–3`.

```
exitflag =
     1
fval =
  2.7048e+002
output =
       iterations: 10
        funcCount: 10
      cgiterations: 15
    firstorderopt: 7.5339e–005
        algorithm: 'large-scale: trust-region reflective Newton'
```

Nonlinear Minimization with Equality Constraints

The large-scale method for `fmincon` can handle equality constraints if no other constraints exist. Suppose you want to minimize the same objective in Eq. 1-7, which is coded in the function `brownfgh.m`, where $n = 1000,$ such that $Aeq \cdot x = beq$ for Aeq that has 100 equations (so Aeq is a 100-by-1000 matrix).

Step 1: Write an M-file brownfgh.m that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix

As before, this file is rather lengthy and is not included here. You can view the code with the command

```
type brownfgh
```

Because `brownfgh` computes the gradient and Hessian values as well as the objective function, you need to use `optimset` to indicate this information is available in `brownfgh` using the `GradObj` and `Hessian` parameters.

The sparse matrix `Aeq` and vector `beq` are available in the file `browneq.mat`

```
load browneq
```

The linear constraint system is 100-by-1000, has unstructured sparsity (use `spy(Aeq)` to view the sparsity structure) and is not too badly ill-conditioned:

```
condest(Aeq*Aeq')
ans =
  2.9310e+006
```

Step 2: Call a nonlinear minimization routine with a starting point xstart

```
fun = 'brownfgh';
load browneq   % get Aeq and beq, the linear equalities
n = 1000;
xstart = –ones(n,1); xstart(2:2:n) = 1;
options = optimset('GradObj','on','Hessian','on', ...
                   'PrecondBandWidth', inf);
[x,fval,exitflag,output] = ...
   fmincon('brownfgh',xstart,[],[],Aeq,beq,[],[],[],options);
```

Setting the parameter `PrecondBandWidth` to `inf` will cause a sparse direct solver to be used instead of preconditioned conjugate gradients.

The `exitflag` indicates convergence with the final function value `fval` after 16 iterations

```
exitflag =
     1
fval =
  205.9313
output =
       iterations: 16
        funcCount: 16
     cgiterations: 14
    firstorderopt: 2.1434e–004
        algorithm: 'large-scale: projected trust-region Newton'
```

The linear equalities are satisfied at `x`

```
norm(Aeq*x-beq)
ans =
  1.1913e–012
```

Quadratic Minimization with Bound Constraints

To minimize a large-scale quadratic with upper and lower bounds, you can use the quadprog function.

The problem stored in the MAT-file qpbox1.mat is a positive definite quadratic, and the Hessian matrix H is tridiagonal, subject to upper (ub) and lower (lb) bounds.

Load the Hessian and define f, lb, ub. Call a quadratic minimization routine with a starting point xstart

```
load qpbox1   % Get H
lb = zeros(400,1); lb(400) = –inf;
ub = 0.9*ones(400,1); ub(400) = inf;
f = zeros(400,1); f([1 400]) = –2;
xstart = 0.5*ones(400,1);
[x,fval,exitflag,output] = ...
        quadprog(H,f,[],[],[],[],lb,ub,xstart);
```

Looking at the resulting values of exitflag and output

```
exitflag =
     1
output =
    firstorderopt: 7.8435e–006
       iterations: 20
     cgiterations: 1809
        algorithm: 'large-scale: reflective trust-region'
```

you can see that while convergence occurred in 20 iterations, the high number of CG iterations indicates that the cost of the linear system solve is high. In light of this cost, one strategy would be to limit the number of CG iterations per optimization iteration. The default number is the dimension of the problem divided by two, 200 for this problem. Suppose you limit it to 50 using the MaxPCGIter flag in options

```
options = optimset('MaxPCGIter',50);
[x,fval,exitflag,output] = ...
        quadprog(H,f,[],[],[],[],lb,ub,xstart,options);
```

This time convergence still occurs and the total number of CG iterations (1547) has dropped.

```
exitflag =
     1
output =
    firstorderopt: 2.3821e-005
       iterations: 36
     cgiterations: 1547
        algorithm: 'large-scale: reflective trust-region'
```

A second strategy would be to use a direct solver at each iteration by setting the `PrecondBandWidth` parameter to `inf`.

```
options = optimset('PrecondBandWidth',inf);
[x,fval,exitflag,output] = ...
        quadprog(H,f,[],[],[],[],lb,ub,xstart,options);
```

Now the number of iterations has dropped to 10.

```
exitflag =
     1
output =
    firstorderopt: 4.8955e-007
       iterations: 10
     cgiterations: 9
        algorithm: 'large-scale: reflective trust-region'
```

Using a direct solve at each iteration usually causes the number of iterations to decrease, but often takes more time per iteration. For this problem, the trade-off is beneficial as the time for `quadprog` to solve the problem decreases by a factor of 10.

Linear Least-Squares with Bound Constraints

Many situations give rise to sparse linear least-squares problems, often with bounds on the variables. The next problem requires that the variables be nonnegative. This problem comes from fitting a function approximation to a piecewise linear spline. Specifically, particles are scattered on the unit square. The function to be approximated is evaluated at these points, and a piecewise linear spline approximation is constructed under the condition that (linear) coefficients are not negative. There are 2000 equations to fit on 400 variables.

```
load particle   % Get C, d
lb = zeros(400,1);
[x,resnorm,residual,exitflag,output] = ...
              lsqlin(C,d,[],[],[],[],lb);
```

The default diagonal preconditioning works fairly well.

```
exitflag =
     1
resnorm =
    22.5794
output =
        algorithm: 'large-scale: trust-region reflective Newton'
    firstorderopt: 2.7870e-005
       iterations: 10
     cgiterations: 42
```

The first-order optimality can be improved (decreased) by using a sparse *QR*-factorization in each iteration: set `PrecondBandWidth` to `inf`.

```
options = optimset('PrecondBandWidth',inf);
[x,resnorm,residual,exitflag,output] = ...
              lsqlin(C,d,[],[],[],[],lb,[],[],options);
```

The number of iterations and the first-order optimality both decrease

```
exitflag =
     1
resnorm =
    22.5794
output =
        algorithm: 'large-scale: trust-region reflective Newton'
    firstorderopt: 5.5907e–015
       iterations: 12
     cgiterations: 11
```

Linear Programming with Equalities and Inequalities

The problem is

$$\min f^T x \qquad \text{such that} \qquad \begin{aligned} Aeq \cdot x &= beq \\ A \cdot x &\le b \\ x &\ge 0 \end{aligned}$$

and you can load the matrices and vectors `A`, `Aeq`, `b`, `beq`, `f` and the lower bounds `lb` into the MATLAB workspace with

```
load sc50b
```

This problem in `sc50b.mat` has 48 variables, 30 inequalities and 20 equalities.

You can use `linprog` to solve the problem.

```
[x,fval,exitflag,output] = ...
    linprog(f,A,b,Aeq,beq,lb,[],[],optimset('Display','iter'));
```

Since the iterative display was set using `optimset`, the results printed to the command line window are

```
  Residuals:   Primal     Dual     Duality    Total
               Infeas    Infeas      Gap       Rel
               A*x-b    A'*y+z-f    x'*z      Error
  ---------------------------------------------------
  Iter    0:  1.50e+003 2.19e+001 1.91e+004 1.00e+002
  Iter    1:  1.15e+002 2.94e-015 3.62e+003 9.90e-001
  Iter    2:  1.16e-012 2.21e-015 4.32e+002 9.48e-001
  Iter    3:  3.23e-012 5.16e-015 7.78e+001 6.88e-001
  Iter    4:  5.78e-011 7.61e-016 2.38e+001 2.69e-001
  Iter    5:  9.31e-011 1.84e-015 5.05e+000 6.89e-002
  Iter    6:  2.96e-011 1.62e-016 1.64e-001 2.34e-003
  Iter    7:  1.51e-011 2.74e-016 1.09e-005 1.55e-007
  Iter    8:  1.51e-012 2.37e-016 1.09e-011 1.51e-013

Optimization terminated successfully.
```

For this problem the large-scale linear programming algorithm quickly reduces the scaled residuals below the default tolerance of `1e-08`.

The `exitflag` value is positive telling you `linprog` converged. You can also get the final function value in `fval` and the number of iterations in `output.iterations`.

```
exitflag =
     1
fval =
 -70.0000
output =
      iterations: 8
    cgiterations: 0
       algorithm: 'lipsol'
```

Linear Programming with Dense Columns in the Equalities

The problem is

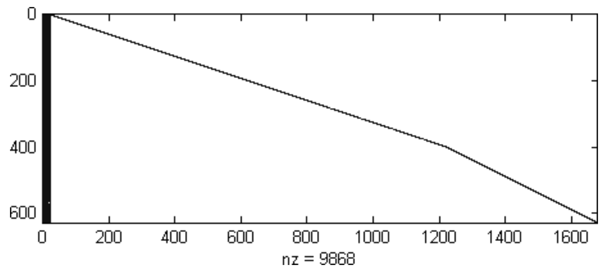
$$\min f^T x \qquad \text{such that} \quad \begin{aligned} Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

and you can load the matrices and vectors `Aeq`, `beq`, `f`, `lb`, and `ub` into the MATLAB workspace with

```
load densecolumns
```

The problem in `densecolumns.mat` has 1677 variables and 627 equalities with lower bounds on all the variables, and upper bounds on 399 of the variables. The equality matrix `Aeq` has dense columns among its first 25 columns, which is easy to see with a spy plot.

```
spy(Aeq)
```



You can use linprog to solve the problem.

```
[x,fval,exitflag,output] = ...
  linprog(f,[],[],Aeq,beq,lb,ub,[],optimset('Display','iter'));
```

Since the iterative display was set using optimset, the results printed to the command line window are

```
  Residuals:   Primal     Dual     Upper    Duality     Total
               Infeas    Infeas    Bounds     Gap        Rel
               A*x-b   A'*y+z-w-f {x}+s-ub  x'*z+s'*w   Error
  -----------------------------------------------------------
  Iter    0:  1.67e+003 8.11e+002 1.35e+003 5.30e+006 2.92e+001
  Iter    1:  1.37e+002 1.33e+002 1.11e+002 1.27e+006 2.48e+000
  Iter    2:  3.56e+001 2.38e+001 2.89e+001 3.42e+005 1.99e+000
  Iter    3:  4.86e+000 8.88e+000 3.94e+000 1.40e+005 1.89e+000
  Iter    4:  4.24e-001 5.89e-001 3.44e-001 1.91e+004 8.41e-001
  Iter    5:  1.23e-001 2.02e-001 9.97e-002 8.41e+003 5.79e-001
  Iter    6:  3.98e-002 7.91e-002 3.23e-002 4.05e+003 3.52e-001
  Iter    7:  7.25e-003 3.83e-002 5.88e-003 1.85e+003 1.85e-001
  Iter    8:  1.47e-003 1.34e-002 1.19e-003 8.12e+002 8.52e-002
  Iter    9:  2.52e-004 3.39e-003 2.04e-004 2.78e+002 2.99e-002
  Iter   10:  3.46e-005 1.08e-003 2.81e-005 1.09e+002 1.18e-002
  Iter   11:  6.95e-007 1.53e-012 5.64e-007 1.48e+001 1.62e-003
  Iter   12:  1.04e-006 2.26e-012 3.18e-008 8.32e-001 9.09e-005
  Iter   13:  3.08e-006 1.23e-012 3.86e-009 7.26e-002 7.94e-006
  Iter   14:  3.75e-007 1.09e-012 6.53e-012 1.11e-003 1.21e-007
  Iter   15:  5.21e-008 1.30e-012 3.27e-013 8.62e-008 9.15e-010

Optimization terminated successfully.
```

You can see the returned values of exitflag, fval, and output .

```
exitflag =
     1
fval =
  9.1464e+003
output =
      iterations: 15
    cgiterations: 225
       algorithm: 'lipsol'
```

This time the number of PCG iterations (in `output.cgiterations`) is nonzero because the dense columns in `Aeq` are detected. Instead of using a sparse Cholesky factorization, `linprog` tries to use the Sherman-Morrison formula to solve a linear system involving `Aeq*Aeq'`. If the Sherman-Morrison formula does not give a satisfactory residual, a PCG iteration is used. See “Main Algorithm” in the “Large-Scale Linear Programming” section of Chapter 3.

Default Parameter Settings

The options structure contains parameters used in the optimization routines. If, on the first call to an optimization routine, the options structure is not provided, or is empty, a set of default parameters is generated.

Some of the default options parameters are calculated using factors based on problem size, such as MaxFunEvals. Some parameters are dependent on the specific optimization routine and are documented in Chapter 4. The parameters in the options structure are shown in Table 4-3 in Chapter 4.

Changing the Default Settings

The function optimset creates or updates an OPTIONS variable to pass to the various optimization functions. The arguments to the optimset function are parameter name and parameter value pairs, such as TolX and 1e–4. Any unspecified properties have default values. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names. For parameter values that are strings, however, case and the exact string are necessary.

help optimset provides information that defines the different parameters and describes how to use them.

Here are some examples of the use of optimset.

Returning All Parameters

optimset returns all the parameters that can be set with typical values and default values.

Determining Parameters Used by a Function

The options structure defines the parameters that can be used by the functions provided by the toolbox. Because functions do not use all the parameters, it may be useful to find which parameters are used by a particular function.

To determine which options structure fields are used by a function, pass the name of the function (in this example, fmincon) to optimset:

```
optimset('fmincon')
```

or

```
optimset fmincon
```

This statement returns a structure. Fields not used by the function have empty values (`[]`); fields used by the function are set to their default values for the given function.

Displaying Output

To display output at each iteration instead of just at termination, enter:

```
options = optimset('Display', 'iter');
```

This command sets the `options.Display` field value to `'iter'`, which causes the toolbox to display output at each iteration.

Running Medium-Scale Optimization

For functions that support medium- and large-scale optimization problems, the default is for the function to use the large-scale algorithm. To use the medium-scale algorithm, enter:

```
options = optimset('LargeScale', 'off');
```

Setting More Than One Parameter

You can specify multiple parameters with one call to `optimset`. For example, to reset the output option and the tolerance on x, enter:

```
options = optimset('Display', 'iter', 'TolX', 1e-6);
```

Updating an options Structure

To update an existing `options` structure, call `optimset` and pass `options` as the first argument:

```
options = optimset(options, 'Display', 'iter', 'TolX', 1e-6);
```

Retrieving Parameter Values

Use the `optimget` function to get parameter values from an `options` structure. For example, to get the current display option, enter:

```
verbosity = optimget(options, 'Display');
```

Displaying Iterative Output

Output Headings: Medium-Scale Algorithms

When the options Display parameter is set to 'iter' for fminsearch, fminbnd, fzero, fgoalattain, fmincon, lsqcurvefit, fminunc, fsolve, lsqnonlin, fminimax, and fseminf, output is produced in column format.

For fminsearch the column headings are

```
Iteration   Func-count     min f(x)         Procedure
```

where

- Iteration is the iteration number.
- Func-count is the number of function evaluations.
- min f(x) is the minimum function value in the current simplex.
- Procedure gives the current simplex operation: initial, expand, reflect, shrink, contract inside and contract outside.

For fzero and fminbnd the column headings are

```
Func-count       x             f(x)         Procedure
```

where

- Func-count is the number of function evaluations (which for fzero is the same as the number of iterations).
- x is the current point.
- f(x) is the current function value at x.
- Procedure gives the current operation. For fzero these include initial (initial point), search (search for a interval containing a zero), bisection (bisection search), and interpolation. For fminbnd, the possible operations are initial, golden (golden section search), and parabolic (parabolic interpolation).

For `fminunc`, the column headings are

```
                                                          Directional
  Iteration  Func-count     f(x)         Step-size     derivative
```

where

- `Iteration` is the iteration number.
- `Func-count` is the number of function evaluations.
- `f(x)` is the current function value.
- `Step-size` is the step size in the current search direction.
- `Directional derivative` is the gradient of the function along the search direction.

For `fsolve`, `lsqnonlin`, and `lsqcurvefit` the headings are

```
                                                   Directional
  Iteration  Func-count  Residual  Step-size   derivative   Lambda
```

where `Iteration`, `Func-count`, `Step-size`, and `Directional derivative` are the same as for `fminunc`, and

- `Residual` is the residual (sum-of-squares) of the function.
- `Lambda` is the λ_k value defined in the "Least Squares Optimization" section of the *Introduction to Algorithms* chapter. (This value is printed when the Levenberg-Marquardt method is used and omitted when the Gauss-Newton method is used.)

For `fmincon` and `fseminf` the headings are

```
                           max                Directional
  Iter   F-count f(x) constraint Step-size derivative Procedure
```

where

- `Iter` is the iteration number.
- `F-count` is the number of function evaluations.
- `f(x)` is the current function value.
- `max constraint` is the maximum constraint violation.
- `Step-size` is the step size in the search direction.

- `Directional derivative` is the gradient of the function along the search direction.
- `Procedures` are messages about the Hessian update and QP subproblem.

The `Procedures` messages are discussed in the "Updating the Hessian Matrix" section of the *Introduction to Algorithms* chapter.

For `fgoalattain` and `fminimax`, the headings are the same as for `fmincon` except `f(x)` and `max constraint` are combined into `Max{F,constraints}` which gives the maximum goal violation or constraint violation for `fgoalattain`, and the maximum function value or constraint violation for `fminimax`.

Output Headings: Large-Scale Algorithms

For `fminunc`, the column headings are

```
                           Norm of      First-order
  Iteration       f(x)       step       optimality   CG-iterations
```

where

- `Iteration` is the iteration number.
- `f(x)` is the current function value.
- `Norm of step` is the norm of the current step-size.
- `First-order Optimality` is the infinity norm of the current gradient.
- `CG-iterations` is the number of iterations taken by PCG (see "Preconditioned Conjugate Gradients" in Chapter 3) at the current (optimization) iteration.

For `lsqnonlin`, `lsqcurvefit`, and `fsolve` the column headings are

```
                             Norm of   First-order
  Iteration  Func-count f(x)   step     optimality CG-iterations
```

where

- `Iteration` is the iteration number.
- `Func-count` is the number of function evaluations.
- `f(x)` is the sum-of-the-squares of the current function values.
- `Norm of step` is the norm of the current step-size.

- `First-order optimality` is a measure of first-order optimality. For bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in “Box Constraints” in Chapter 3 and `g` is the gradient. For unconstrained problems, it is the infinity norm of the current gradient.
- `CG-iterations` is the number of iterations taken by PCG (see “Preconditioned Conjugate Gradients” in Chapter 3) at the current (optimization) iteration.

For `fmincon`, the column headings are

```
                               Norm of      First-order
 Iteration        f(x)          step        optimality  CG-iterations
```

where

- `Iteration` is the iteration number.
- `f(x)` is the current function value.
- `Norm of step` is the norm of the current step-size.
- `First-order optimality` is a measure of first-order optimality. For bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in “Box Constraints” in Chapter 3 and `g` is the gradient. For equality constrained problems, it is the infinity norm of the projected gradient. (The projected gradient is the gradient projected into the nullspace of `Aeq`.)
- `CG-iterations` is the number of iterations taken by PCG (see “Preconditioned Conjugate Gradients” in Chapter 3) at the current (optimization) iteration.

For `linprog` the column headings are

```
    Residuals:    Primal      Dual      Upper    Duality     Total
                  Infeas     Infeas     Bounds     Gap        Rel
                  A*x—b   A'*y+z—w—f {x}+s—ub  x'*z+s'*w    Error
```

where

- `Primal Infeas A*x-b` is the norm of the residual `A*x — b`.
- `Dual Infeas A'*y+z—w—f` is the norm of the residual `A'*y+z—w—f` (where `w` is all zero if there are no finite upper bounds).

- `Upper Bounds x'*z+s'*w` is the norm of the residual `spones(s).*x+s–ub` (which is defined to be zero if all variables are unbounded above). This column is not printed if no finite upper bounds exist.
- `Duality Gap x'*z+s'*w` is the duality gap (see "Large-Scale Linear Programming" in Chapter 3) between the primal objective and the dual objective. `s` and `w` only appear in this equation if there are finite upper bounds.
- `Total Rel Error` is the total relative error described at the end of the "Main Algorithm" subsection of the section "Large-Scale Linear Programming" in Chapter 3.

Optimization of Inline Objects Instead of M-Files

The routines in the Optimization Toolbox also perform optimization on inline objects, avoiding the need to write M-files to define functions.

To represent a mathematical function at the command line, create an `inline` object from a string expression. For example, you can create an inline object of the `humps` function (use the command `type humps` to see the M-file function `humps.m`):

```
f = inline('1./((x–0.3).^2 + 0.01) + 1./((x–0.9).^2 + 0.04)–6');
```

You can then evaluate `f` at 2.0:

```
f(2.0)
ans =
   –4.8552
```

And you can pass `f` to an optimization routine to minimize it:

```
x = fminbnd(f, 3, 4)
```

You can also create functions of more than one argument with `inline` by specifying the names of the input arguments along with the string expression. For example, to use `lsqcurvefit`, you need a function that takes two input arguments, `x` and `xdata`:

```
f= inline('sin(x).*xdata +(x.^2).*cos(xdata)','x','xdata')
x = pi; xdata = pi*[4;2;3];
f(x, xdata)
ans =
  9.8696e+000
  9.8696e+000
 –9.8696e+000
```

and then call `lsqcurvefit`:

```
% assume ydata exists
x = lsqcurvefit(f,x,xdata,ydata)
```

Other examples using this technique follow.

A matrix equation

```
x = fsolve(inline('x∗x∗x–[1,2;3,4]'),ones(2,2))
```

A nonlinear least squares problem

```
x = lsqnonlin(inline('x*x–[3 5;9 10]'),eye(2,2))
```

Finally, another example using `fgoalattain` where the function has additional arguments to pass to the optimization routine. For example, if the function to be minimized has additional arguments `A`, `B`, and `C`,

```
fun = inline('sort(eig(A+B*x*C))','x','A','B','C');
x = fgoalattain(fun,–ones(2,2),[–5,–3,–1],[5, 3, 1],...
[ ],[ ],[ ],[ ],–4*ones(2),4*ones(2),[ ],[ ],A,B,C);
```

solves the problem detailed in Chapter 4 for `fgoalattain`.

Practicalities

Optimization problems can take many iterations to converge and can be sensitive to numerical problems such as truncation and round-off error in the calculation of finite-difference gradients. Most optimization problems benefit from good starting guesses. This improves the execution efficiency and can help locate the global minimum instead of a local minimum.

Advanced problems are best solved by an evolutionary approach whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

The Optimization Toolbox functions can be applied to a large variety of problems. Used with a little "conventional wisdom," many of the limitations associated with optimization techniques can be overcome. Additionally, problems that are not typically in the standard form can be handled by using an appropriate transformation. Below is a list of typical problems and recommendations for dealing with them:

Problem

The solution does not appear to be a global minimum.

Recommendation

There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.

Problem

The `fminunc` function produces warning messages and seems to exhibit slow convergence near the solution.

Recommendation

If you are not supplying analytically determined gradients and the termination criteria are stringent, `fminunc` often exhibits slow convergence near the solution due to truncation error in the gradient calculation. Relaxing the termination criteria produces faster, although less accurate, solutions. For the medium-scale algorithm, another option is adjusting the finite-difference perturbation levels, `DiffMinChange` and `DiffMaxChange`, which may increase the accuracy of gradient calculations.

Problem

Sometimes an optimization problem has values of `x` for which it is impossible to evaluate the objective function `fun` or the nonlinear constraints `nonlcon`.

Recommendation

Place bounds on the independent variables or make a penalty function to give a large positive value to `f` and `g` when infeasibility is encountered. For gradient calculation the penalty function should be smooth and continuous.

Problem

The function that is being minimized has discontinuities.

Recommendation

The derivation of the underlying method is based upon functions with continuous first and second derivatives. Some success may be achieved for some classes of discontinuities when they do not occur near solution points. One option is to smooth the function. For example, the objective function might include a call to an interpolation function to do the smoothing.

Or, for the medium-scale algorithms, the finite-difference parameters may be adjusted in order to jump over small discontinuities. The variables `DiffMinChange` and `DiffMaxChange` control the perturbation levels for `x` used in the calculation of finite-difference gradients. The perturbation, Δ `x`, is always in the range

```
DiffMinChange < Dx < DiffMaxChange
```

Problem

Warning messages are displayed.

Recommendation

This sometimes occurs when termination criteria are overly stringent, or when the problem is particularly sensitive to changes in the independent variables. This usually indicates truncation or round-off errors in the finite-difference gradient calculation, or problems in the polynomial interpolation routines. These warnings can usually be ignored because the routines continue to make steps toward the solution point; however, they are often an indication that convergence will take longer than normal. Scaling can sometimes improve the sensitivity of a problem.

Problem

The independent variables, `x`, only can take on discrete values, for example, integers.

Recommendation

This type of problem occurs commonly when, for example, the variables are the coefficients of a filter that are realized using finite-precision arithmetic or when the independent variables represent materials that are manufactured only in standard amounts.

Although the Optimization Toolbox functions are not explicitly set up to solve discrete problems, some discrete problems can be solved by first solving an equivalent continuous problem. Discrete variables can be progressively eliminated from the independent variables, which are free to vary.

Eliminate a discrete variable by rounding it up or down to the nearest best discrete value. After eliminating a discrete variable, solve a reduced order problem for the remaining free variables. Having found the solution to the reduced order problem, eliminate another discrete variable and repeat the cycle until all the discrete variables have been eliminated.

`dfildemo` is a demonstration routine that shows how filters with fixed precision coefficients can be designed using this technique.

Problem

The minimization routine appears to enter an infinite loop or returns a solution that does not satisfy the problem constraints.

Recommendation

Your objective, constraint or gradient functions may be returning Inf, NaN, or complex values. The minimization routines expect only real numbers to be returned. Any other values may cause unexpected results. Insert some checking code into the user-supplied functions to verify that only real numbers are returned (use the function `isfinite`).

Problem

You do not get the convergence you expect from the `lsqnonlin` routine.

Recommendation

You may be forming the sum of squares explicitly and returning a scalar value. `lsqnonlin` expects a vector (or matrix) of function values that are squared and summed internally.

Converting Your Code to Version 2.0 Syntax

Most of the function names and calling sequences have changed in Version 2 to accommodate new functionality and to clarify the roles of the input and output variables.

As a result, if you want to use the new versions of these functions, you need to modify any code that currently uses the old function names and calling sequences.

This table lists the functions provided by the toolbox and indicates the functions whose names have changed in Version 2.

Old (Version 1.5) Name	New (Version 2) Name
`attgoal`	`fgoalattain`
`conls`	`lsqlin`
`constr`	`fmincon`
`curvefit`	`lsqcurvefit`
`fmin`	`fminbnd`
`fmins`	`fminsearch`
`fminu`	`fminunc`
`fsolve`	`fsolve` (name unchanged)
`fzero`	`fzero` (name unchanged)
`leastsq`	`lsqnonlin`
`minimax`	`fminimax`
`nnls`	`lsqnonneg`
`lp`	`linprog`
`qp`	`quadprog`
`seminf`	`fseminf`

This section explains the reasons for the new calling sequences and explains how to convert your code. In addition, it provides a detailed example of rewriting a call to the `constr` function to call the new `fmincon` function instead.

In addition to the information in this section, consult the M-file help for the new functions for more information about the arguments they take. For example, to see the help for `fmincon`, type:

```
help fmincon
```

Using optimset and optimget

The `optimset` function replaces `foptions` for overriding default parameter settings. See "Changing the Default Settings" in Chapter 1 for more information on using `optimset` and `optimget`.

New Calling Sequences

Version 2 of the toolbox makes these changes in the calling sequences:

- Equality constraints and inequality constraints are now supplied as separate input arguments.
- Linear constraints and nonlinear constraints are now supplied as separate input arguments.
- The gradient of the objective is computed in the same function as the objective, rather than in a separate function, in order to provide more efficient computation (because the gradient and objective often share similar computations). Similarly, the gradient of the nonlinear constraints is computed by the (now separate) nonlinear constraint function.
- The Hessian matrix can be provided by the objective function. (This matrix is used only by the new large-scale algorithms.)
- Flags are *required* to indicate when extra information is available:
  - `OPTIONS.GradObj = 'on'` indicates the user-supplied gradient of the objective function is available.
  - `OPTIONS.GradConstr = 'on'` indicates the user-supplied gradient of the constraints is available.
  - `OPTIONS.Hessian = 'on'` indicates the user-supplied Hessian of the objective function is available.

- Each function takes an `OPTIONS` structure to adjust parameters to the optimization functions (see `optimset`, `optimget`).
- The new default output gives information upon termination (the old default was no output, the new default is `OPTIONS.display = 'final'`).
- Each function returns an `EXITFLAG` that denotes the termination state.
- The default uses the new large-scale methods when possible. If you want to use the older algorithms (referred to as medium-scale algorithms in other parts of this *User's Guide*), set `OPTIONS.LargeScale = 'off'`.

Algorithm terminating conditions have been fine tuned. The stopping conditions relating to `TolX` and `TolFun` for the large-scale *and* medium-scale code are joined using OR instead of AND for these functions: `fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, and `lsqnonlin`. As a result, you may need to specify stricter tolerances; the defaults reflect this change.

Each function now has an `OUTPUT` structure that contains information about the problem solution relevant to that function.

The `LAMBDA` is now a structure where each field is the Lagrange multipliers for a type of constraint. For more information, see the individual functions entries in Chapter 4.

The sections below describe how to convert from the old function names and calling sequences to the new ones. The calls shown are the most general cases, involving all possible input and output arguments. Note that many of these arguments are optional; see the online help for these functions for more information.

Converting from attgoal to fgoalattain

In Version 1.5, you used this call to `attgoal`:

```
OPTIONS = foptions;
[X,OPTIONS] = attgoal('FUN',x0,GOAL, WEIGHT, OPTIONS, VLB, VUB,
   'GRADFUN', P1, P2,...);
```

with `[F] = FUN(X,P1,...)` and `[DF] = GRADFUN(X,P1,...)`.

In Version 2, you call `fgoalattain` like this:

```
OPTIONS = optimset('fgoalattain');
[X,FVAL,ATTAINFACTOR,EXITFLAG,OUTPUT,LAMBDA] =
   fgoalattain('FUN',x0,GOAL,WEIGHT,A,B,Aeq,Beq,VLB,VUB,
   'NONLCON',OPTIONS,P1,P2,...);
```

with `[F,DF] = FUN(X,P1,P2,...)` and `NONLCON = []`.

The `fgoalattain` function now allows nonlinear constraints, so you can now define:

```
[Cineq,Ceq,DCineq,DCeq] = NONLCON(X,P1,...)
```

Converting from conls to lsqlin

In Version 1.5, you used this call to `conls`:

```
[X,LAMBDA,HOW] = conls(A,b,C,d,VLB,VUB,X0,N,DISPLAY);
```

In Version 2, convert the input arguments to the correct form for `lsqlin` by separating the equality and inequality constraints:

```
Ceq = C(1:N,:);
deq = d(1:N);
C = C(N+1:end,:);
d = d(N+1:end,:);
```

Now call `lsqlin` like this:

```
OPTIONS = optimset('Display','final');
[X,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA] =
   lsqlin(A,b,C,d,Ceq,deq,VLB,VUB,X0,OPTIONS);
```

Converting from constr to fmincon

In Version 1.5, you used this call to `constr`:

```
[X,OPTIONS,LAMBDA,HESS] =
   constr('FUN',x0,OPTIONS,VLB,VUB,'GRADFUN',P1,P2,...);
```

with `[F,C] = FUN(X,P1,...)` and `[G,DC] = GRADFUN(X,P1,...)`.

In Version 2, replace `FUN` and `GRADFUN` with two new functions:

- `OBJFUN`, which returns the objective function, the gradient (first derivative) of this function, and its Hessian matrix (second derivative):

  ```
  [F,G,H] = OBJFUN(X,P1,...)
  ```

- `NONLCON`, which returns the functions for the nonlinear constraints (both inequality and equality constraints) and their gradients:

  ```
  [C,Ceq,DC,DCeq] = NONLCON(X,P1,...)
  ```

Now call `fmincon` like this:

```
% OBJFUN supplies the objective gradient and Hessian;
% NONLCON supplies the constraint gradient.
OPTIONS = 
   optimset('GradObj','on','GradConstr','on','Hessian','on');
[X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD,HESSIAN] = 
   fmincon('OBJFUN',x0,A,B,Aeq,Beq,VLB,VUB,'NONLCON',OPTIONS,
   P1,P2,...);
```

See “Example of Converting from constr to fmincon” for a detailed example of converting from `constr` to `fmincon`.

Converting from curvefit to lsqcurvefit

In Version 1.5, you used this call to `curvefit`:

```
[X,OPTIONS,FVAL,JACOBIAN] = 
   curvefit('FUN',x0,XDATA,YDATA,OPTIONS,'GRADFUN',P1,P2,...);
```

with `F = FUN(X,P1,...)` and `G = GRADFUN(X,P1,...)`.

In Version 2, replace `FUN` and `GRADFUN` with a single function that returns both `F` and `G` (the objective function and the gradient):

```
[F,G] = OBJFUN(X,P1,...)
```

Now call `lsqcurvefit` like this:

```
OPTIONS = optimset('GradObj','on');    % Gradient is supplied
VLB = []; VUB = [];                 % New arguments not in curvefit
[X,RESNORM,F,EXITFLAG,OUTPUT,LAMBDA,JACOB] = 
   lsqcurvefit('OBJFUN',x0,XDATA,YDATA,VLB,VUB,OPTIONS,
   P1,P2,...);
```

If you have an existing FUN and GRADFUN that you do not want to rewrite, you can pass them both to lsqcurvefit by placing them in a cell array:

```
OPTIONS = optimset('GradObj','on');    % Gradient is supplied
VLB = []; VUB = [];                 % New arguments not in curvefit
[X,RESNORM,F,EXITFLAG,OUTPUT,LAMBDA,JACOB] =
   lsqcurvefit({'FUN','GRADFUN'},x0,XDATA,YDATA,VLB,VUB,
   OPTIONS,P1,P2,...);
```

Converting from fmin to fminbnd

In Version 1.5, you used this call to fmin:

```
[X,OPTIONS] = fmin('FUN',x1,x2,OPTIONS,P1,P2,...);
```

In Version 2, you call fminbnd like this:

```
[X,FVAL,EXITFLAG,OUTPUT] = fminbnd(FUN,x1,x2,OPTIONS,P1,P2,...);
```

Converting from fmins to fminsearch

In Version 1.5, you used this call to fmins:

```
[X,OPTIONS] = fmins('FUN',x0,OPTIONS,[],P1,P2,...);
```

In Version 2, you call fminsearch like this:

```
[X,FVAL,EXITFLAG,OUTPUT] = fminsearch(FUN,x0,OPTIONS,P1,P2,...);
```

Converting from fminu to fminunc

In Version 1.5, you used this call to fminu:

```
[X,OPTIONS] = fminu('FUN',x0,OPTIONS,'GRADFUN',P1,P2,...);
```

with `F = FUN(X,P1, ...)` and `G = GRADFUN(X,P1, ...)`.

In Version 2, replace FUN and GRADFUN with a single function that returns both F and G (the objective function and the gradient):

```
[F,G] = OBJFUN(X,P1, ...)
```

(This function can also return the Hessian matrix as a third output argument.)

Now call `fminunc` like this:

```
OPTIONS = optimset('GradObj','on');       % Gradient is supplied
[X,FVAL,EXITFLAG,OUTPUT,GRAD,HESSIAN] =
   fminunc('OBJFUN',x0,OPTIONS,P1,P2,...);
```

If you have an existing `FUN` and `GRADFUN` that you do not want to rewrite, you can pass them both to `fminunc` by placing them in a cell array:

```
OPTIONS = optimset('GradObj','on');       % Gradient is supplied
[X,FVAL,EXITFLAG,OUTPUT,GRAD,HESSIAN] =
   fminunc({'FUN','GRADFUN'},x0,OPTIONS,P1,P2,...);
```

Converting to the new form of fsolve

In Version 1.5, you used this call to `fsolve`:

```
[X,OPTIONS] =
   fsolve('FUN',x0,XDATA,YDATA,OPTIONS,'GRADFUN',P1,P2,...);
```

with `F = FUN(X,P1,...)` and `G = GRADFUN(X,P1,...)`.

In Version 2, replace `FUN` and `GRADFUN` with a single function that returns both `F` and `G` (the objective function and the gradient):

```
[F,G] = OBJFUN(X,P1, ...)
```

Now call `fsolve` like this:

```
OPTIONS = optimset('GradObj','on');       % Gradient is supplied
[X,FVAL,EXITFLAG,OUTPUT,JACOBIAN] =
   fsolve('OBJFUN',x0,OPTIONS,P1,P2,...);
```

If you have an existing `FUN` and `GRADFUN` that you do not want to rewrite, you can pass them both to `fsolve` by placing them in a cell array:

```
OPTIONS = optimset('GradObj','on');       % Gradient is supplied
[X,FVAL,EXITFLAG,OUTPUT,JACOBIAN] =
   fsolve({'FUN','GRADFUN'},x0,OPTIONS,P1,P2,...);
```

Converting to the new form of fzero

In Version 1.5, you used this call to `fzero`:

```
X = fzero(F,X,TOL,TRACE,P1,P2,...);
```

In Version 2, replace the TRACE and TOL arguments with:

```
if TRACE == 0,
   val = 'none';
elseif TRACE == 1
   val = 'iter';
end
OPTIONS = optimset('Display',val,'TolX',TOL);
```

Now call fzero like this:

```
[X,FVAL,EXITFLAG,OUTPUT] = fzero(F,X,OPTIONS,P1,P2,...);
```

Converting from leastsq to lsqnonlin

In Version 1.5, you used this call to leastsq:

```
[X,OPTIONS,FVAL,JACOBIAN] =
   leastsq('FUN',x0,OPTIONS,'GRADFUN',P1,P2,...);
```

with F = FUN(X,P1,...) and G = GRADFUN(X,P1, ...).

In Version 2, replace FUN and GRADFUN with a single function that returns both F and G (the objective function and the gradient):

```
[F,G] = OBJFUN(X,P1, ...)
```

Now call lsqnonlin like this:

```
OPTIONS = optimset('GradObj','on');   % Gradient is supplied
VLB = []; VUB = [];                  % New arguments not in leastsq
[X,RESNORM,F,EXITFLAG,OUTPUT,LAMBDA,JACOBIAN] =
   lsqnonlin('OBJFUN',x0,VLB,VUB,OPTIONS,P1,P2,...);
```

If you have an existing FUN and GRADFUN that you do not want to rewrite, you can pass them both to lsqnonlin by placing them in a cell array:

```
OPTIONS = optimset('GradObj','on');   % Gradient is supplied
VLB = []; VUB = [];                  % New arguments not in leastsq
[X,RESNORM,F,EXITFLAG,OUTPUT,LAMBDA,JACOBIAN] =
   lsqnonlin({'FUN','GRADFUN'},x0,VLB,VUB,OPTIONS,P1,P2,...);
```

Converting from lp to linprog

In Version 1.5, you used this call to `lp`:

```
[X,LAMBDA,HOW] = lp(f,A,b,VLB,VUB,X0,N,DISPLAY);
```

In Version 2, convert the input arguments to the correct form for `linprog` by separating the equality and inequality constraints:

```
Aeq = A(1:N,:);
beq = b(1:N);
A = A(N+1:end,:);
b = b(N+1:end,:);
if DISPLAY
   val = 'final';
else
   val = 'none';
end
OPTIONS = optimset('Display',val);
```

Now call `linprog` like this:

```
[X,FVAL,EXITFLAG,OUTPUT,LAMBDA] =
   linprog(f,A,b,Aeq,beq,VLB,VUB,X0,OPTIONS);
```

Converting from minimax to fminimax

In Version 1.5, you used this call to `minimax`:

```
[X,OPTIONS] =
   minimax('FUN',x0,OPTIONS,VLB,VUB,'GRADFUN',P1,P2,...);
```

with `F = FUN(X,P1,...)` and `G = GRADFUN(X,P1,...)`.

In Version 2, you call `fminimax` like this:

```
OPTIONS = optimset('fminimax');
[X,FVAL,MAXFVAL,EXITFLAG,OUTPUT,LAMBDA] =
   fminimax('OBJFUN',x0,A,B,Aeq,Beq,VLB,VUB,'NONLCON',OPTIONS,
   P1,P2,...);
```

with `[F,DF] = OBJFUN(X,P1,...)`

and `[Cineq,Ceq,DCineq,DCeq] = NONLCON(X,P1,...)`.

Converting from nnls to lsqnonneg

In Version 1.5, you used this call to `nnls`:

```
[X,LAMBDA] = nnls(A,b,tol);
```

In Version 2, replace the `tol` argument with:

```
OPTIONS = optimset('Display','none','TolX',tol);
```

Now call `lsqnonneg` like this:

```
[X,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA] =
   lsqnonneg(A,b,X0,OPTIONS);
```

Converting from qp to quadprog

In Version 1.5, you used this call to `qp`:

```
[X,LAMBDA,HOW] = qp(H,f,A,b,VLB,VUB,X0,N,DISPLAY);
```

In Version 2, convert the input arguments to the correct form for `quadprog` by separating the equality and inequality constraints:

```
Aeq = A(1:N,:);
beq = b(1:N);
A = A(N+1:end,:);
b = b(N+1:end,:);
if DISPLAY
   val = 'final';
else
   val = 'none';
end
OPTIONS = optimset('Display',val);
```

Now call `quadprog` like this:

```
[X,FVAL,EXITFLAG,OUTPUT,LAMBDA] =
   quadprog(H,f,A,b,Aeq,beq,VLB,VUB,X0,OPTIONS);
```

Converting from seminf to fseminf

In Version 1.5, you used this call to `seminf`:

```
[X,OPTIONS] = seminf('FUN',N,x0,OPTIONS,VLB,VUB,P1,P2,...);
```

with `[F,C,PHI1,PHI2,...,PHIN,S] = FUN(X,S,P1,P2,...)`.

In Version 2, call `fseminf` like this:

```
[X,FVAL,EXITFLAG,OUTPUT,LAMBDA] =
   fseminf('OBJFUN',x0,N,'NONLCON',A,B,Aeq,Beq,VLB,VUB,OPTIONS,
   P1,P2,...);
```

with `F = OBJFUN(X,P1,...)`

and `[Cineq,Ceq,PHI1,PHI2,...,PHIN,S] = NONLCON(X,S,P1,...)`.

Example of Converting from constr to fmincon

Old Call to constr

```
OPTIONS = foptions;
OPTIONS(13) = 2;  % two equality constraints
OPTIONS(1) = 1;
OPTIONS(9) = 1;
A1 = [ 1 4 –3]; b1 = 2;
A2 = [ 2 5 0]; b2 = 9;
x0 = [1; .5; .8];
LB = []; UB = [];
[X,OPTIONS,LAMBDA,HESS] = ...
   constr('myfuncon',x0,OPTIONS,LB,UB,'mygradcon',A1,b1,A2,b2);

% myfuncon.m
[F, C] = myfuncon(x,A1,b1,A2,b2)
F = x(1) + 0.0009*x(2)^3 + sin(x(3));

C(1,1) = A1*x–b;            % equality linear constraint
C(2,1) = 3*x(1)^2–1;        % equality nonlinear constraint

C(3,1) = A2*x–b2;           % inequality linear constraint
C(4,1) = 7*sin(x(2))–1;     % inequality nonlinear constraint

% mygradcon.m
[G, DC] = mygradcon(x,alpha)
G = [1;                      % gradient of the objective
     3*0.0009*x(2)^2;
     cos(x(3))];

DC(:,1) = A1';               % gradient of the constraints
DC(:,2) = [6*x(1); 0; 0];
DC(:,3) = A2';
DC(:,4) = [0; 7*cos(x(2)); 0];
```

New Call to fmincon

```
OPTIONS = optimset(...
   'Display', 'iter', ...
   'GradCheck', 'on', ... % Check gradients.
   'GradObj', 'on', ...   % Gradient of objective is provided.
   'GradConstr', 'on');   % Gradient of constraints is provided.

A1 = [ 1 4 -3]; b1 = 2;   % linear equalities
A2 = [ 2 5 0]; b2 = 9;    % linear inequalities

x0 = [1; .5; .8];
LB = []; UB = [];
[X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD,HESSIAN] = ...
   fmincon('myfun',x0,A2,b2,A1,b1,LB,UB,'mycon',OPTIONS);

% myfun.m
function [F,G] = myfun(x)
F = x(1) + 0.0009*x(2)^3 + sin(x(3));
G = [1;
     3*0.0009*x(2)^2;
     cos(x(3))];

% mycon.m
function [C,Ceq,DC,DCeq]= mycon(x)
Ceq(1,1) = 3*x(1)^2-1;        % equality nonlinear constraint
C(1,1) = 7*sin(x(2))-1;       % inequality nonlinear constraint
DCeq(:,1) = [6*x(1); 0; 0];   % gradient of equality
                              %   nonlinear constraint
DC(:,1) = [0; 7*cos(x(2)); 0]; % gradient of inequality
                              %   nonlinear constraint
```

Introduction to Algorithms

Parametric Optimization

This chapter provides an introduction to the different optimization problem formulations and describes the “medium-scale” algorithms used in the Optimization Toolbox. “Medium-scale” is not a standard term and is used here only to differentiate these algorithms from the large-scale algorithms described in the *Large-scale Algorithms* chapter.

Parametric optimization is used to find a set of design parameters, $x = \{x_1,x_2,\ldots,x_n\}$, that can in some way be defined as optimal. In a simple case this may be the minimization or maximization of some system characteristic that is dependent on *x*. In a more advanced formulation the objective function, *f(x)*, to be minimized or maximized, may be subject to constraints in the form of equality constraints, $G_i(x) = 0\ (i = 1,\ldots,m_e)$, inequality constraints, $G_i(x) \le 0\ (i = m_e + 1,\ldots,m)$, and/or parameter bounds, x_l, x_u.

A General Problem (**GP**) description is stated as

$$\begin{aligned}
&\underset{x \in \Re^n}{\text{minimize}} \quad f(x) \\
&\text{subject to:} \quad G_i x = 0, \qquad i = 1,\ldots,m_e \\
&\qquad\qquad\qquad G_i(x) \le 0, \qquad i = m_e + 1,\ldots,m \\
&\qquad\qquad\qquad x_l \le x \le x_u
\end{aligned} \tag{2-1}$$

where *x* is the vector of design parameters, $(x \in \Re^n)$, *f(x)* is the objective function that returns a scalar value ($f(x)\colon\ \Re^n \to \Re$), and the vector function *G(x)* returns the values of the equality and inequality constraints evaluated at *x* ($G(x)\colon\ \Re^n \to \Re^m$).

An efficient and accurate solution to this problem is not only dependent on the size of the problem in terms of the number of constraints and design variables but also on characteristics of the objective function and constraints. When both the objective function and the constraints are linear functions of the design variable, the problem is known as a Linear Programming problem (LP). Quadratic Programming (QP) concerns the minimization or maximization of a quadratic objective function that is linearly constrained. For both the LP and QP problems, reliable solution procedures are readily available. More difficult to solve is the Nonlinear Programming (NP) problem in which the objective function and constraints may be nonlinear functions of the design variables. A solution of the NP problem generally requires an iterative procedure to establish a direction of search at each major iteration. This is usually achieved by the solution of an LP, a QP, or an unconstrained sub-problem.

Unconstrained Optimization

Although a wide spectrum of methods exists for unconstrained optimization, methods can be broadly categorized in terms of the derivative information that is, or is not, used. Search methods that use only function evaluations (e.g., the simplex search of Nelder and Mead [33]) are most suitable for problems that are very nonlinear or have a number of discontinuities. Gradient methods are generally more efficient when the function to be minimized is continuous in its first derivative. Higher order methods, such as Newton's method, are only really suitable when the second order information is readily and easily calculated since calculation of second order information, using numerical differentiation, is computationally expensive.

Gradient methods use information about the slope of the function to dictate a direction of search where the minimum is thought to lie. The simplest of these is the method of steepest descent in which a search is performed in a direction, $-\nabla f(x)$, (where $\nabla f(x)$ is the gradient of the objective function). This method is very inefficient when the function to be minimized has long narrow valleys as, for example, is the case for Rosenbrock's function

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

The minimum of this function is at $x = [1,1]$ where $f(x) = 0$. A contour map of this function is shown in Figure , along with the solution path to the minimum for a steepest descent implementation starting at the point [–1.9,2]. The optimization was terminated after 1000 iterations, still a considerable distance from the minimum. The black areas are where the method is continually zig-zagging from one side of the valley to another. Note that towards the center of the plot, a number of larger steps are taken when a point lands exactly at the center of the valley.

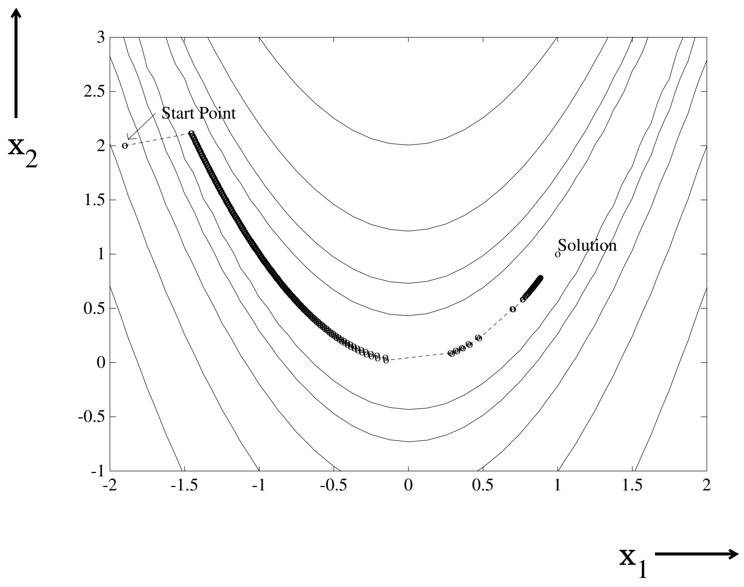


Figure 2-1: Steepest Descent Method on Rosenbrock's Function (Eq. 2-2)

This type of function (Eq. 2-2), also known as the banana function, is notorious in unconstrained examples because of the way the curvature bends around the origin. Eq. 2-2 is used throughout this section to illustrate the use of a variety of optimization techniques. The contours have been plotted in exponential increments due to the steepness of the slope surrounding the U-shaped valley.

Quasi-Newton Methods

Of the methods that use gradient information, the most favored are the quasi-Newton methods. These methods build up curvature information at each iteration to formulate a quadratic model problem of the form

$$\min_{x} \frac{1}{2}x^T Hx + c^T x + b \tag{2-3}$$

where the Hessian matrix, H, is a positive definite symmetric matrix, c is a constant vector, and b is a constant. The optimal solution for this problem occurs when the partial derivatives of x go to zero, i.e.,

$$\nabla f(x^*) = Hx^* + c = 0 \tag{2-4}$$

The optimal solution point, x^*, can be written as

$$x^* = -H^{-1}c \tag{2-5}$$

Newton-type methods (as opposed to quasi-Newton methods) calculate H directly and proceed in a direction of descent using a line search method to locate the minimum after a number of iterations. Calculating H numerically involves a large amount of computation. Quasi-Newton methods avoid this by using the observed behavior of $f(x)$ and $\nabla f(x)$ to build up curvature information to make an approximation to H using an appropriate updating technique.

A large number of Hessian updating methods have been developed. Generally, the formula of Broyden [3], Fletcher [4], Goldfarb [5], and Shanno [6] (BFGS) is thought to be the most effective for use in a general purpose method.

The formula is given by

BFGS

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T s_k^T s_k H_k}{s_k^T H_k s_k} \tag{2-6}$$

where

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

As a starting point, H_0 can be set to any symmetric positive definite matrix, for example, the identity matrix I. To avoid the inversion of the Hessian H, you can derive an updating method in which the direct inversion of H is avoided by using a formula that makes an approximation of the inverse Hessian H^{-1} at each update. A well known procedure is the DFP formula of Davidon [7], Fletcher, and Powell [8]. This uses the same formula as the above BFGS method (Eq.) except that q_k is substituted for s_k.

The gradient information is either supplied through analytically calculated gradients, or derived by partial derivatives using a numerical differentiation method via finite differences. This involves perturbing each of the design variables, x, in turn and calculating the rate of change in the objective function.

At each major iteration, k, a line search is performed in the direction

$$d = -H_k^{-1} \cdot \nabla f(x_k) \tag{2-7}$$

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function (Eq. 2-2) in Figure 2-2. The method is able to follow the shape of the valley and converges to the minimum after 140 function evaluations using only finite difference gradients.

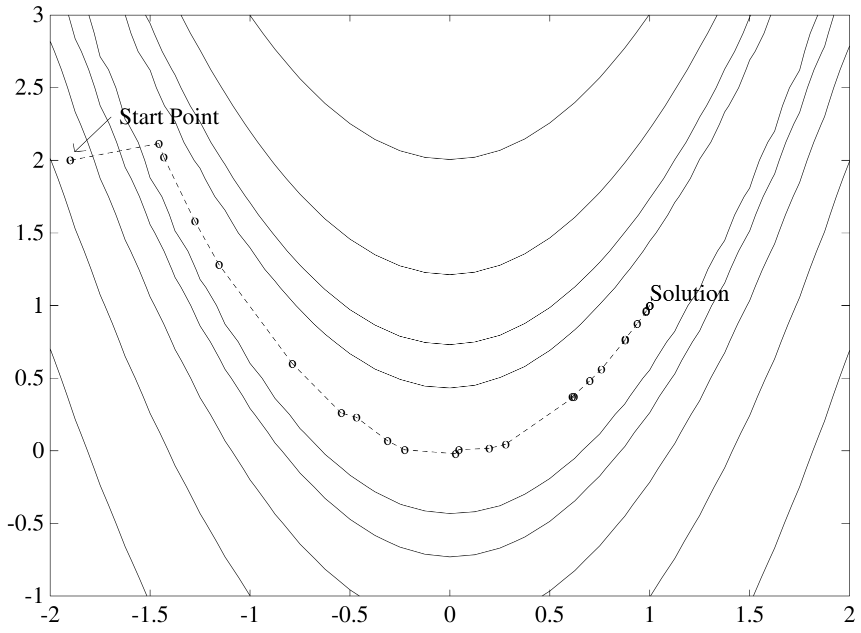


Figure 2-2: BFGS Method on Rosenbrock's Function

Line Search

Most unconstrained and constrained methods use the solution of a sub-problem to yield a search direction in which the solution is estimated to lie. The minimum along the line formed from this search direction is generally approximated using a search procedure (e.g., Fibonacci, Golden Section) or by a polynomial method involving interpolation or extrapolation (e.g., quadratic, cubic). Polynomial methods approximate a number of points with a univariate polynomial whose minimum can be calculated easily. Interpolation refers to the condition that the minimum is bracketed (i.e., the minimum lies in the area spanned by the available points), whereas extrapolation refers to a minimum located outside the range spanned by the available points. Extrapolation methods are generally considered unreliable for estimating minima for nonlinear functions. However, they are useful for estimating step length when trying to bracket the minimum as shown in the "Line Search Procedures" section. Polynomial interpolation methods are generally the most effective in terms of efficiency when the function to be minimized is continuous. The problem is to find a new iterate x_{k+1} of the form

$$x_{k+1} = x_k + \alpha^* d \qquad \textbf{(2-8)}$$

where x_k denotes the current iterate, d the search direction obtained by an appropriate method, and α^* is a scalar step length parameter that is the distance to the minimum.

Quadratic Interpolation

Quadratic interpolation involves a data fit to a univariate function of the form

$$m_q(\alpha) = a\alpha^2 + b\alpha + c \qquad \textbf{(2-9)}$$

where an extremum occurs at a step length of

$$\alpha^* = \frac{-b}{2a} \qquad \textbf{(2-10)}$$

This point may be a minimum or a maximum. It is a minimum when interpolation is performed (i.e., using a bracketed minimum) or when a is positive. Determination of coefficients, a and b, can be found using any combination of three gradient or function evaluations. It may also be carried out with just two gradient evaluations. The coefficients are determined through the formulation and solution of a linear set of simultaneous equations.

Various simplifications in the solution of these equations can be achieved when particular characteristics of the points are used. For example, the first point can generally be taken as $\alpha = 0$. Other simplifications can be achieved when the points are evenly spaced. A general problem formula is as follows:

Given three unevenly spaced points $\{x_1, x_2, x_3\}$ and their associated function values $\{f(x_1), f(x_2), f(x_3)\}$ the minimum resulting from a second-order fit is given by

Quadratic Interpolation

$$x_{k+1} = \frac{1}{2} \frac{\beta_{23} f(x_1) + \beta_{31} f(x_2) + \beta_{12} f(x_3)}{\gamma_{23} f(x_1) + \gamma_{31} f(x_2) + \gamma_{12} f(x_3)} \qquad \textbf{(2-11)}$$

where

$$\beta_{ij} = x_i^2 - x_j^2$$

$$\gamma_{ij} = x_i - x_j$$

For interpolation to be performed, as opposed to extrapolation, the minimum must be bracketed so that the points can be arranged to give

$$f(x_2) < f(x_1) \qquad \text{and} \qquad f(x_2) < f(x_3)$$

Cubic Interpolation

Cubic interpolation is useful when gradient information is readily available or when more than three function evaluations have been calculated. It involves a data fit to the univariate function

$$m_c(\alpha) = a\alpha^3 + b\alpha^2 + c\alpha + d \qquad \textbf{(2-12)}$$

where the local extrema are roots of the quadratic equation

$$3a\alpha^2 + 2b\alpha + c = 0$$

To find the minimum extremum, take the root that gives $6a\alpha + 2b$ as positive. Coefficients a and b can be determined using any combination of four gradient or function evaluations, or alternatively, with just three gradient evaluations.

The coefficients are calculated by the formulation and solution of a linear set of simultaneous equations. A general formula, given two points, $\{x_1, x_2\}$, their corresponding gradients with respect to x, $\{\nabla f(x_1), \nabla f(x_2)\}$, and associated function values, $\{f(x_1), f(x_2)\}$ is

$$x_{k+1} = x_2 - (x_2 - x_1)\frac{\nabla f(x_2) + \beta_2 - \beta_1}{\nabla f(x_2) - \nabla f(x_1) + 2\beta_2} \tag{2-13}$$

where

$$\beta_1 = \nabla f(x_1) + \nabla f(x_2) - 3\frac{f(x_1) - f(x_2)}{x_1 - x_2}$$

$$\beta_2 = (\beta_1^2 - \nabla f x_1 \nabla f(x_2))^{1/2}.$$

Quasi-Newton Implementation

A quasi-Newton algorithm is used in `fminunc`. The algorithm consists of two phases:

- Determination of a direction of search
- Line search procedure

Implementation details of the two phases are discussed below.

Hessian Update

The direction of search is determined by a choice of either the BFGS (Eq.) or the DFP method given in the "Quasi-Newton Methods" section (set the `options` parameter `HessUpdate` to `'dfp'` to select the DFP method). The Hessian, H, is always maintained to be positive definite so that the direction of search, d, is always in a descent direction. This means that for some arbitrarily small step, α, in the direction, d, the objective function decreases in magnitude. Positive definiteness of H is achieved by ensuring that H is initialized to be positive definite and thereafter $q_k^T s_k$ (from Eq.) is always positive. The term $q_k^T s_k$ is a product of the line search step length parameter, α_k and a combination of the search direction, d, with past and present gradient evaluations,

$$q_k^T s_k = \alpha_k(\nabla f(x_{k+1})^T d - \nabla f(x_k)^T d) \qquad \textbf{(2-14)}$$

The condition that $q_k^T s_k$ is positive is always achieved by ensuring that a sufficiently accurate line search is performed. This is because the search direction, d, is a descent direction so that α_k and $-\nabla f(x_k)^T d$ are always positive. Thus, the possible negative term $\nabla f(x_{k+1})^T d$ can be made as small in magnitude as required by increasing the accuracy of the line search.

Line Search Procedures

Two line search strategies are used depending on whether gradient information is readily available or whether it must be calculated using a finite difference method. When gradient information is available, the default is to use a cubic polynomial method. When gradient information is not available, the default is to use a mixed quadratic and cubic polynomial method.

Cubic Polynomial Method

In the proposed cubic polynomial method, a gradient and a function evaluation is made at every iteration, k. At each iteration an update is performed when a new point is found, x_{k+1}, which satisfies the condition that

$$f(x_{k+1}) < f(x_k) \tag{2-15}$$

At each iteration a step, α_k, is attempted to form a new iterate of the form

$$x_{k+1} = x_k + \alpha_k d \tag{2-16}$$

If this step does not satisfy the condition (Eq. 2-15) then α_k is reduced to form a new step, α_{k+1}. The usual method for this reduction is to use bisection (i.e., to continually halve the step length until a reduction is achieved in *f(x)*. However, this procedure is slow when compared to an approach that involves using gradient and function evaluations together with cubic interpolation/extrapolation methods to identify estimates of step length.

When a point is found that satisfies the condition (Eq. 2-15), an update is performed if $q_k^T s_k$ is positive. If it is not, then further cubic interpolations are performed until the univariate gradient term $\nabla f(x_{k+1})^T d$ is sufficiently small so that $q_k^T s_k$ is positive.

It is usual practice to reset α_k to unity after every iteration. However, note that the quadratic model (Eq. 2-3) is generally only a good one near to the solution point. Therefore, α_k, is modified at each major iteration to compensate for the case when the approximation to the Hessian is monotonically increasing or decreasing. To ensure that, as x_k approaches the solution point, the procedure reverts to a value of α_k close to unity, the values of $q_k^T s_k - \nabla f(x_k)^T d$ and α_{k+1} are used to estimate the closeness to the solution point and thus to control the variation in α_k.

After each update procedure, a step length α_k is attempted, following which a number of scenarios are possible. Consideration of all the possible cases is quite complicated and so they are represented pictorially in Figure 2-3, where the left-hand point on the graphs represents the point x_k. The slope of the line bisecting each point represents the slope of the univariate gradient, $\nabla f(x_k)^T d$, which is always negative for the left-hand point. The right-hand point is the point x_{k+1} after a step of α_k is taken in the direction d.

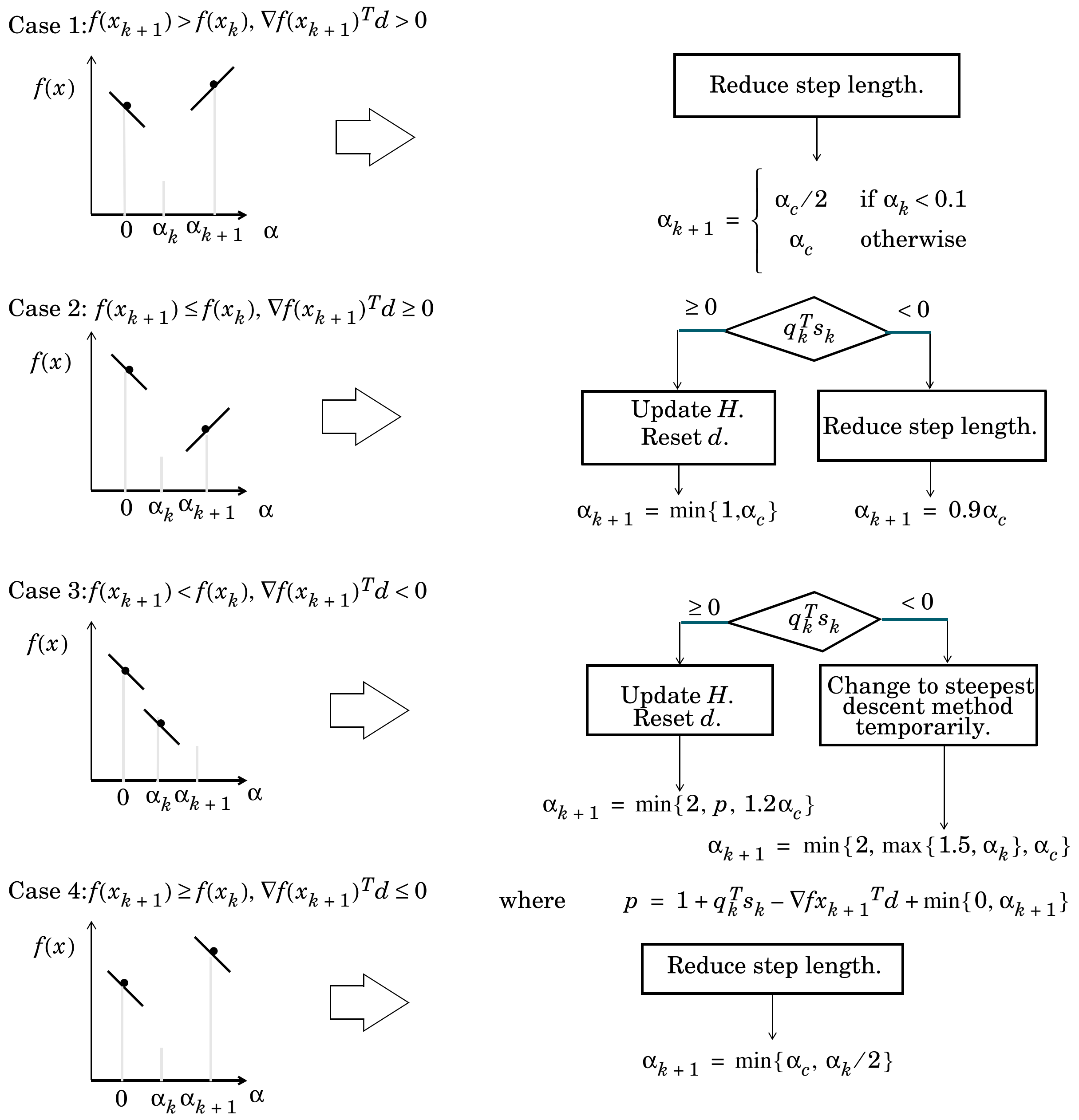

Case 1: $f(x_{k+1}) > f(x_k)$, $\nabla f(x_{k+1})^T d > 0$
$f(x)$
0 α_k α_{k+1} α
Reduce step length.
$\alpha_{k+1} = \begin{cases} \alpha_c/2 & \text{if } \alpha_k < 0.1 \\ \alpha_c & \text{otherwise} \end{cases}$
Case 2: $f(x_{k+1}) \le f(x_k)$, $\nabla f(x_{k+1})^T d \ge 0$
$f(x)$
0 α_k α_{k+1} α
$q_k^T s_k$
≥ 0
< 0
Update H. Reset d.
Reduce step length.
$\alpha_{k+1} = \min\{1, \alpha_c\}$
$\alpha_{k+1} = 0.9\alpha_c$
Case 3: $f(x_{k+1}) < f(x_k)$, $\nabla f(x_{k+1})^T d < 0$
$f(x)$
0 α_k α_{k+1} α
$q_k^T s_k$
≥ 0
< 0
Update H. Reset d.
Change to steepest descent method temporarily.
$\alpha_{k+1} = \min\{2, p, 1.2\alpha_c\}$
$\alpha_{k+1} = \min\{2, \max\{1.5, \alpha_k\}, \alpha_c\}$
where $p = 1 + q_k^T s_k - \nabla f x_{k+1}{}^T d + \min\{0, \alpha_{k+1}\}$
Case 4: $f(x_{k+1}) \ge f(x_k)$, $\nabla f(x_{k+1})^T d \le 0$
$f(x)$
0 α_k α_{k+1} α
Reduce step length.
$\alpha_{k+1} = \min\{\alpha_c, \alpha_k/2\}$

Figure 2-3: Cubic Polynomial Line Search Procedures

Cases 1 and 2 show the procedures performed when the value $\nabla f(x_{k+1})^T d$ is positive. Cases 3 and 4 show the procedures performed when the value $\nabla f(x_{k+1})^T d$ is negative. The notation $\min\{a, b, c\}$ refers to the smallest value of the set $\{a, b, c\}$.

At each iteration a cubicly interpolated step length α_c is calculated and then used to adjust the step length parameter α_{k+1}. Occasionally, for very nonlinear functions α_c may be negative, in which case α_c is given a value of $2\alpha_k$. The methods for changing the step length have been refined over a period of time by considering a large number of test problems.

Certain robustness measures have also been included so that, even in the case when false gradient information is supplied, a reduction in *f(x)* can be achieved by taking a negative step. This is done by setting $\alpha_{k+1} = -\alpha_k/2$ when α_k falls below a certain threshold value (e.g., 1e–8). This is important when extremely high precision is required if only finite difference gradients are available.

Mixed Cubic/Quadratic Polynomial Method

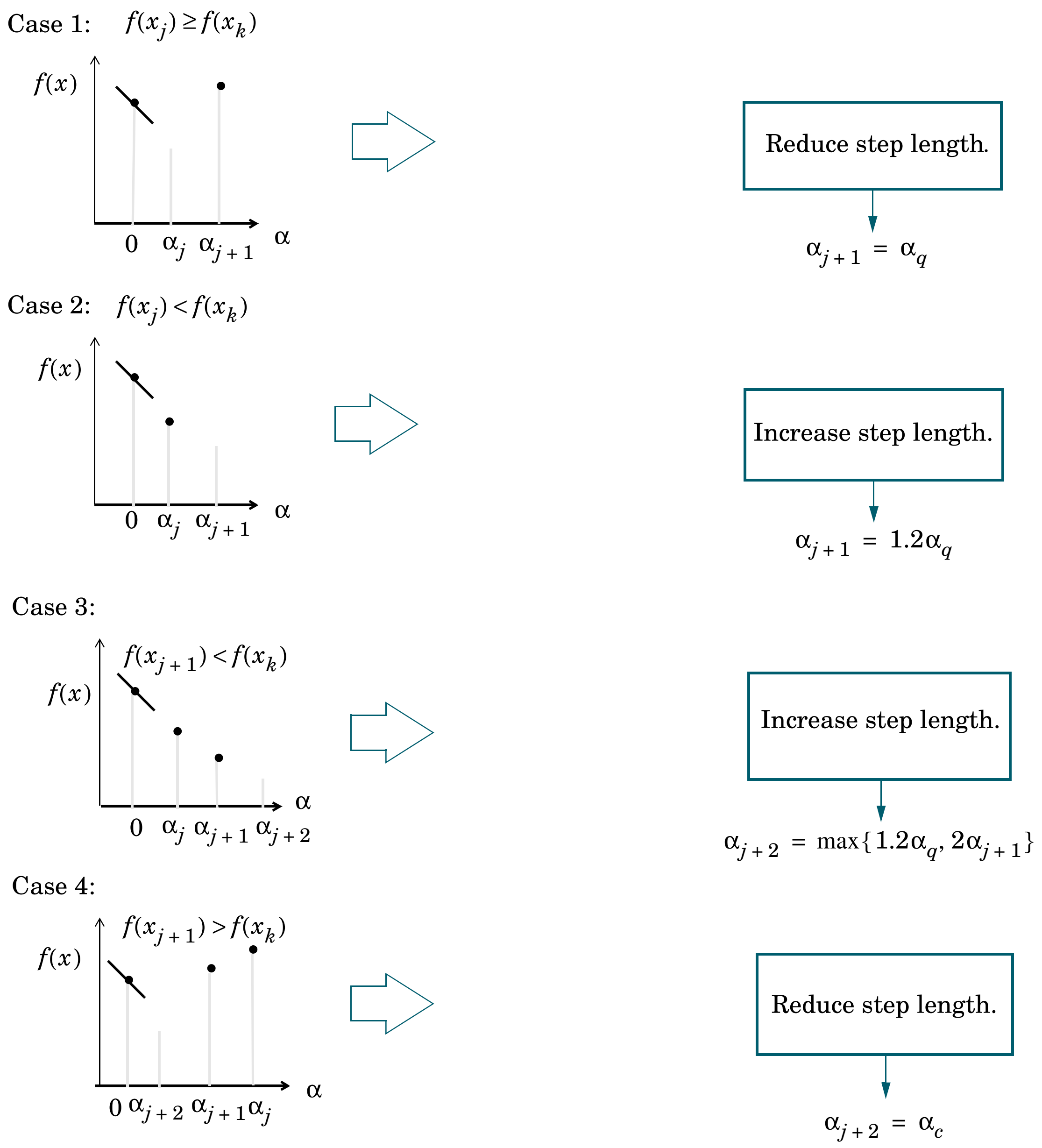
The cubic interpolation/extrapolation method has proved successful for a large number of optimization problems. However, when analytic derivatives are not available, the evaluating finite difference gradients is computationally expensive. Therefore, another interpolation/extrapolation method is implemented so that gradients are not needed at every iteration. The approach in these circumstances, when gradients are not readily available, is to use a quadratic interpolation method. The minimum is generally bracketed using some form of bisection method. This method, however, has the disadvantage that all the available information about the function is not used. For instance, a gradient calculation is always performed at each major iteration for the Hessian update. Therefore, given three points that bracket the minimum, it is possible to use cubic interpolation, which is likely to be more accurate than using quadratic interpolation. Further efficiencies are possible if, instead of using bisection to bracket the minimum, extrapolation methods similar to those used in the cubic polynomial method are used.

Hence, the method that is used in `fminunc`, `lsqnonlin`, `lsqcurvefit`, and `fsolve` is to find three points that bracket the minimum and to use cubic interpolation to estimate the minimum at each line search. The estimation of step length, at each minor iteration, j, is shown in Figure for a number of point combinations. The left-hand point in each graph represents the function value $f(x_1)$ and univariate gradient $\nabla f(x_k)$ obtained at the last update. The right-hand points represent the points accumulated in the minor iterations of the line search procedure.

The terms α_q and α_c refer to the minimum obtained from a respective quadratic and cubic interpolation or extrapolation. For highly nonlinear functions, α_c and α_q may be negative, in which case they are set to a value of $2\alpha_k$ so that they are always maintained to be positive. Cases 1 and 2 use quadratic interpolation with two points and one gradient to estimate a third point that brackets the minimum. If this fails, cases 3 and 4 represent the possibilities for changing the step length when at least three points are available.

When the minimum is finally bracketed, cubic interpolation is achieved using one gradient and three function evaluations. If the interpolated point is greater than any of the three used for the interpolation, then it is replaced with the point with the smallest function value. Following the line search procedure the Hessian update procedure is performed as for the cubic polynomial line search method.

.



Case 1: $f(x_j) \geq f(x_k)$
$f(x)$
0 α_j α_{j+1} α
Reduce step length.
$\alpha_{j+1} = \alpha_q$
Case 2: $f(x_j) < f(x_k)$
$f(x)$
0 α_j α_{j+1} α
Increase step length.
$\alpha_{j+1} = 1.2\alpha_q$
Case 3:
$f(x_{j+1}) < f(x_k)$
$f(x)$
0 α_j α_{j+1} α_{j+2} α
Increase step length.
$\alpha_{j+2} = \max\{1.2\alpha_q, 2\alpha_{j+1}\}$
Case 4:
$f(x_{j+1}) > f(x_k)$
$f(x)$
0 α_{j+2} α_{j+1} α_j α
Reduce step length.
$\alpha_{j+2} = \alpha_c$

Figure 2-4: Line Search Procedures with Only Gradient for the First Point

Least Squares Optimization

The line search procedures used in conjunction with a quasi-Newton method are used in the function `fminunc`. They are also used as part of a nonlinear least squares (LS) optimization routines, `lsqnonlin` and `lsqcurvefit`. In the least squares problem a function, *f(x)* is minimized that is a sum of squares.

LS

$$\min_{x \in \Re^n} f(x) = \frac{1}{2}\| F(x) \|_2^2 = \frac{1}{2}\sum_i F_i(x)^2 \qquad \textbf{(2-17)}$$

Problems of this type occur in a large number of practical applications especially when fitting model functions to data, i.e., nonlinear parameter estimation. They are also prevalent in control where you want the output, $y(x, t)$ to follow some continuous model trajectory, $\phi(t)$, for vector x and scalar t. This problem can be expressed as

$$\min_{x \in \Re^n} \int_{t_2}^{t_1} (y(x,t) - \phi(t))^2 dt \qquad \textbf{(2-18)}$$

where $y(x, t)$ and $\phi(t)$ are scalar functions.

When the integral is discretized using a suitable quadrature formula, Eq. 2-18 can be formulated as a least squares problem

$$\min_{x \in \Re^n} f(x) = \sum_{i=1}^{m} (\bar{y}(x, t_i) - \bar{\phi}(t_i))^2 \qquad \textbf{(2-19)}$$

where y and $\bar{\phi}$ include the weights of the quadrature scheme. Note that in this problem the vector *F(x)* is

$$F(x) = \begin{bmatrix} \bar{y}(x, t_1) - \bar{\phi}(t_1) \\ \bar{y}(x, t_2) - \bar{\phi}(t_2) \\ \cdots \\ \bar{y}(x, t_m) - \bar{\phi}(t_m) \end{bmatrix}$$

In problems of this kind the residual $\| F(x) \|$ is likely to be small at the optimum since it is general practice to set realistically achievable target trajectories. Although the function in LS (Eq. 2-18) can be minimized using a general unconstrained minimization technique as described in the "Unconstrained Optimization" section, certain characteristics of the problem can often be exploited to improve the iterative efficiency of the solution procedure. The gradient and Hessian matrix of LS (Eq. 2-18) have a special structure.

Denoting the $m \times n$ Jacobian matrix of *F(x)* as *J(x)*, the gradient vector of *f(x)* as $G(x)$, the Hessian matrix of *f(x)* as $H(x)$, and the Hessian matrix of each $F_i(x)$ as $H_i(x)$, we have

$$\begin{aligned} G(x) &= 2J(x)^T F(x) \\ H(x) &= 2J(x)^T J(x) + 2Q(x) \end{aligned} \qquad \textbf{(2-20)}$$

where

$$Q(x) = \sum_{i=1}^{m} F_i(x) \cdot H_i(x)$$

The matrix *Q(x)* has the property that when the residual $\| F(x) \|$ tends to zero as x_k approaches the solution, then *Q(x)* also tends to zero. Thus when $\| F(x) \|$ is small at the solution, a very effective method is to use the Gauss-Newton direction as a basis for an optimization procedure.

Gauss-Newton Method

In the Gauss-Newton method, a search direction, d_k, is obtained at each major iteration, *k,* that is a solution of the linear least-squares problem

Gauss-Newton

$$\min_{x \in \Re^n} \| J(x_k) d_k - F(x_k) \|_2^2 \qquad \textbf{(2-21)}$$

The direction derived from this method is equivalent to the Newton direction when the terms of *Q(x)* can be ignored. The search direction d_k can be used as part of a line search strategy to ensure that at each iteration the function *f(x)* decreases.

To consider the efficiencies that are possible with the Gauss-Newton method, Figure shows the path to the minimum on Rosenbrock's function (Eq. 2-2) when posed as a least squares problem. The Gauss-Newton method converges after only 48 function evaluations using finite difference gradients compared to 140 iterations using an unconstrained BFGS method.

The Gauss-Newton method often encounters problems when the second order term *Q(x)* in Eq. 2-20 is significant. A method that overcomes this problem is the Levenberg-Marquardt method.

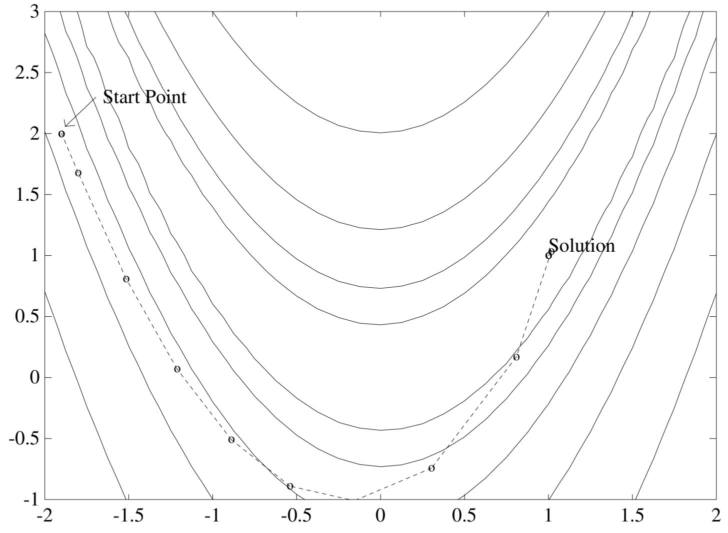


Figure 2-5: Gauss-Newton Method on Rosenbrock's Function

Levenberg-Marquardt Method

The Levenberg-Marquardt [18,19] method uses a search direction that is a solution of the linear set of equations

$$(J(x_k)^T J(x) + \lambda_k I) d_k = -J(x_k) F(x_k) \qquad \textbf{(2-22)}$$

where the scalar λ_k controls both the magnitude and direction of d_k. When λ_k is zero, the direction d_k is identical to that of the Gauss-Newton method. As λ_k tends to infinity, d_k tends towards a vector of zeros and a steepest descent direction. This implies that for some sufficiently large λ_k, the term $F(x_k + d_k) < F(x_k)$ holds true. The term λ_k can therefore be controlled to ensure descent even when second order terms, which restrict the efficiency of the Gauss-Newton method, are encountered.

The Levenberg-Marquardt method therefore uses a search direction that is a cross between the Gauss-Newton direction and the steepest descent. This is illustrated in Figure 2-6 below. The solution for Rosenbrock's function (Eq. 2-2) converges after 90 function evaluations compared to 48 for the Gauss-Newton method. The poorer efficiency is partly because the Gauss-Newton method is generally more effective when the residual is zero at the solution. However, such information is not always available beforehand, and occasional poorer efficiency of the Levenberg-Marquardt method is compensated for by its increased robustness.

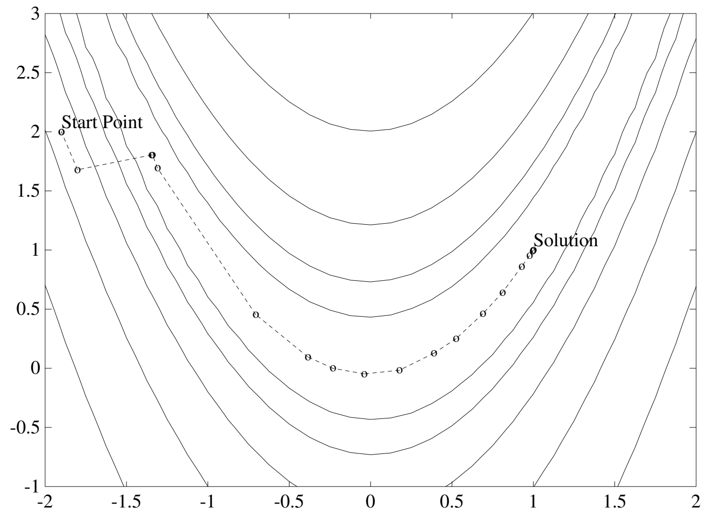


Figure 2-6: Levenberg-Marquardt Method on Rosenbrock's Function

Nonlinear Least Squares Implementation

For a general survey of nonlinear least squares methods see Dennis [21]. Specific details on the Levenberg-Marquardt method can be found in Moré [20]. Both the Gauss-Newton method and the Levenberg-Marquardt method are implemented in the Optimization Toolbox. Details of the implementations are discussed below.

Gauss-Newton Implementation

The Gauss-Newton method is implemented using similar polynomial line search strategies discussed for unconstrained optimization. In solving the linear least squares problem (Prob. 2.18), exacerbation of the conditioning of the equations is avoided by using the QR decomposition of $J(x_k)$ and applying the decomposition to $F(x_k)$ (using the MATLAB \ operator). This is in contrast to inverting the explicit matrix, $J(x_k)^T J(x_k)$, which can cause unnecessary errors to occur.

Robustness measures are included in the method. These measures consist of changing the algorithm to the Levenberg-Marquardt method when either the step length goes below a threshold value (in this implementation 1e–15) or when the condition number of $J(x_k)$ is below 1e–10. The condition number is a ratio of the largest singular value to the smallest.

Levenberg-Marquardt Implementation

The main difficulty in the implementation of the Levenberg-Marquardt method is an effective strategy for controlling the size of λ_k at each iteration so that it is efficient for a broad spectrum of problems. The method used in this implementation is to estimate the relative nonlinearity of *f(x)* using a linear predicted sum of squares $f_p(x_k)$ and a cubicly interpolated estimate of the minimum $f_k(x_*)$. In this way the size of λ_k is determined at each iteration.

The linear predicted sum of squares is calculated as

$$f_p(x_k) = (J(x_{k-1}))^T d_{k-1} + F(x) \tag{2-23}$$

and the term $f_k(x_*)$ is obtained by cubicly interpolating the points $f(x_k)$ and $f(x_{k-1})$. A step length parameter α^* is also obtained from this interpolation, which is the estimated step to the minimum. If $f_p(x_k)$ is greater than $f_k(x_*)$, then λ_k is reduced, otherwise it is increased. The justification for this is that the difference between $f_p(x_k)$ and $f_k(x_*)$ is a measure of the effectiveness of the Gauss-Newton method and the linearity of the problem. This determines whether to use a direction approaching the steepest descent direction or the Gauss-Newton direction. The formulas for the reduction and increase in λ_k, which have been developed through consideration of a large number of test problems, are shown in Figure 2-7 below.

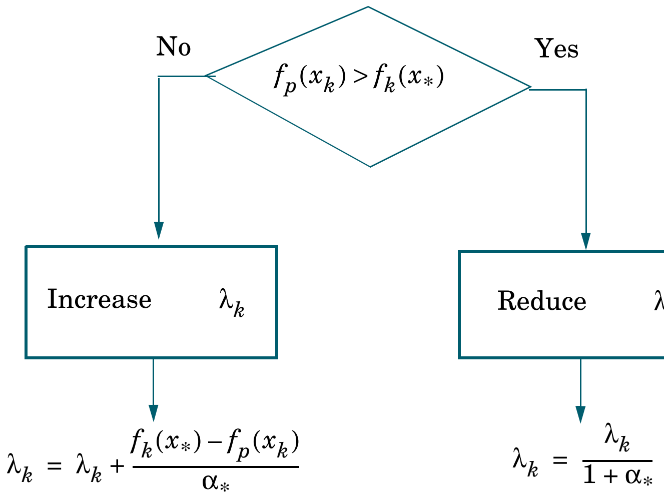

No
$f_p(x_k) > f_k(x_*)$
Yes
Increase λ_k
Reduce λ_k
$\lambda_k = \lambda_k + \frac{f_k(x_*) - f_p(x_k)}{\alpha_*}$
$\lambda_k = \frac{\lambda_k}{1+\alpha_*}$

Figure 2-7: Updating λ_k

Following the update of λ_k, a solution of Eq. 2-22 is used to obtain a search direction, d_k. A step length of unity is then taken in the direction d_k, which is followed by a line search procedure similar to that discussed for the unconstrained implementation. The line search procedure ensures that $f(x_{k+1}) < f(x_k)$ at each major iteration and the method is therefore a descent method.

The implementation has been successfully tested on a large number of nonlinear problems. It has proved to be more robust than the Gauss-Newton method and iteratively more efficient than an unconstrained method. The Levenberg-Marquardt algorithm is the default method used by `lsqnonlin`. The Gauss-Newton method can be selected by setting the `options` parameter `LevenbergMarquardt` to `'off'`.

Constrained Optimization

In constrained optimization, the general aim is to transform the problem into an easier subproblem that can then be solved and used as the basis of an iterative process. A characteristic of a large class of early methods is the translation of the constrained problem to a basic unconstrained problem by using a penalty function for constraints, which are near or beyond the constraint boundary. In this way the constrained problem is solved using a sequence of parameterized unconstrained optimizations, which in the limit (of the sequence) converge to the constrained problem. These methods are now considered relatively inefficient and have been replaced by methods that have focused on the solution of the Kuhn-Tucker (KT) equations. The KT equations are necessary conditions for optimality for a constrained optimization problem. If the problem is a so-called convex programming problem, that is, *f(x)* and $G_i(x), i = 1, \ldots, m$, are convex functions, then the KT equations are both necessary and sufficient for a global solution point.

Referring to GP (Eq. 2-1), the Kuhn-Tucker equations can be stated as

$$
\begin{aligned}
& f(x^*) + \sum_{i=1}^{m} \lambda_i^* \cdot \nabla G_i(x^*) = 0 \\
& \nabla G_i(x^*) = 0 \qquad i = 1, \ldots, m_e \\
& \lambda_i^* \geq 0 \qquad i = m_e + 1, \ldots, m
\end{aligned}
\tag{2-24}
$$

The first equation describes a canceling of the gradients between the objective function and the active constraints at the solution point. For the gradients to be canceled, Lagrange Multipliers ($\lambda_i,\ i = 1, \ldots m$) are necessary to balance the deviations in magnitude of the objective function and constraint gradients. Since only active constraints are included in this canceling operation, constraints that are not active must not be included in this operation and so are given Lagrange multipliers equal to zero. This is stated implicitly in the last two equations of Eq. 2-24.

The solution of the KT equations forms the basis to many nonlinear programming algorithms. These algorithms attempt to compute directly the Lagrange multipliers. Constrained quasi-Newton methods guarantee superlinear convergence by accumulating second order information regarding the KT equations using a quasi-Newton updating procedure. These methods are commonly referred to as Sequential Quadratic Programming (SQP) methods since a QP sub-problem is solved at each major iteration (also known as Iterative Quadratic Programming, Recursive Quadratic Programming, and Constrained Variable Metric methods).

Sequential Quadratic Programming (SQP)

SQP methods represent state-of-the-art in nonlinear programming methods. Schittowski [22], for example, has implemented and tested a version that out performs every other tested method in terms of efficiency, accuracy, and percentage of successful solutions, over a large number of test problems.

Based on the work of Biggs [9], Han [10], and Powell [11,12], the method allows you to closely mimic Newton's method for constrained optimization just as is done for unconstrained optimization. At each major iteration an approximation is made of the Hessian of the Lagrangian function using a quasi-Newton updating method. This is then used to generate a QP sub-problem whose solution is used to form a search direction for a line search procedure. An overview of SQP is found in Fletcher [2], Gill et al. [1], Powell [13], and Schittowski [14]. The general method, however, is stated here.

Given the problem description in GP (Eq. 2.1) the principal idea is the formulation of a QP sub-problem based on a quadratic approximation of the Lagrangian function.

$$L(x,\lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g_i(x) \tag{2-25}$$

Here Eq. 2.1 is simplified by assuming that bound constraints have been expressed as inequality constraints. The QP sub-problem is obtained by linearizing the nonlinear constraints.

QP Subproblem

$$\begin{aligned} &\underset{d \in \Re^n}{\text{minimize}} \quad \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d \\ &\nabla g_i(x_k)^T d + g_i(x_k) = 0 \qquad i = 1, \dots m_e \\ &\nabla g_i(x_k)^T d + g_i(x_k) \le 0 \qquad i = m_e + 1, \dots m \end{aligned} \tag{2-26}$$

This sub-problem can be solved using any QP algorithm (see, for instance, the "Quadratic Programming Solution" section). The solution is used to form a new iterate$x_{k+1} = x_k + \alpha_k d_k$

The step length parameter α_k is determined by an appropriate line search procedure so that a sufficient decrease in a merit function is obtained (see the "Updating the Hessian Matrix" section). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function (Eq. 2-25). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see the section "Updating the Hessian Matrix") appears to be the most popular.

A nonlinearly constrained problem can often be solved in fewer iterations than an unconstrained problem using SQP. One of the reasons for this is that, because of limits on the feasible area, the optimizer can make well-informed decisions regarding directions of search and step length.

Consider Rosenbrock's function (Eq. 2-2) with an additional nonlinear inequality constraint, *g(x)*

$$x_1^2 + x_2^2 - 1.5 \le 0 \qquad \textbf{(2-27)}$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. Figure shows the path to the solution point $x = [0.9072,0.8228]$ starting at $x = [-1.9,2]$.

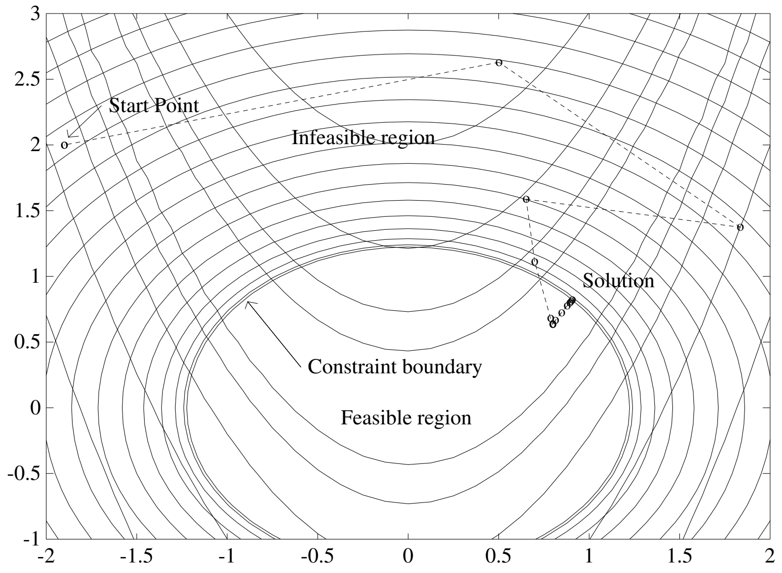


Figure 2-8: SQP Method on Nonlinear Linearly Constrained Rosenbrock's Function (Eq.2-2)

SQP Implementation

The MATLAB SQP implementation consists of three main stages, which are discussed briefly in the following sub-sections:

- Updating of the Hessian matrix of the Lagrangian function
- Quadratic programming problem solution
- Line search and merit function calculation

Updating the Hessian Matrix

At each major iteration a positive definite quasi-Newton approximation of the Hessian of the Lagrangian function, H,is calculated using the BFGS method where λ_i (i = 1,...,m) is an estimate of the Lagrange multipliers.

Hessian Update (BFGS)

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T H_k}{s_k^T H_k s_k} \qquad \text{where}$$

$$s_k = x_{k+1} - x_k \tag{2-28}$$

$$q_k = \nabla f(x_{k+1}) + \sum_{i=1}^{n} \lambda_i \cdot \nabla g_i(x_{k+1}) - \left(\nabla f(x_k) + \sum_{i=1}^{n} \lambda_i \cdot \nabla g_i(x_k) \right)$$

Powell [11] recommends keeping the Hessian positive definite even though it may be positive indefinite at the solution point. A positive definite Hessian is maintained providing $q_k^T s_k$ is positive at each update and that H is initialized with a positive definite matrix. When $q_k^T s_k$ is not positive, q_k is modified on an element by element basis so that $q_k^T s_k > 0$. The general aim of this modification is to distort the elements of q_k, which contribute to a positive definite update, as little as possible. Therefore, in the initial phase of the modification, the most negative element of $q_k.*s_k$ is repeatedly halved. This procedure is continued until $q_k^T s_k$ is greater than or equal to 1e-5. If after this procedure, $q_k^T s_k$ is still not positive, q_k is modified by adding a vector v multiplied by a constant scalar w, that is,

$$q_k = q_k + wv \tag{2-29}$$

$$v_i = \nabla g_i(x_{k+1}) \cdot g_i(x_{k+1}) - \nabla g_i(x_k) \cdot g_i(x_k),$$

where $\quad$ if $(q_k)_i \cdot w < 0$ and $(q_k)_i \cdot (s_k)_i < 0 \quad (i = 1, \ldots m)$

$$v_i = 0 \quad \text{otherwise}$$

and w is systematically increased until $q_k^T s_k$ becomes positive.

The functions `fmincon`, `fminimax`, `fgoalattain`, and `fseminf` all use SQP. If the `options` parameter `Display` is set to `'iter'`, then various information is given such as function values and the maximum constraint violation. When the Hessian has to be modified using the first phase of the procedure described above to keep it positive definite, then `Hessian modified` is displayed. If the Hessian has to be modified again using the second phase of the approach described above, then `Hessian modified twice` is displayed. When the QP sub-problem is infeasible, then infeasible will be displayed. Such displays are usually not a cause for concern but indicate that the problem is highly nonlinear and that convergence may take longer than usual. Sometimes the message `no update` is displayed indicating that $q_k^T s_k$ is nearly zero. This can be an indication that the problem setup is wrong or you are trying to minimize a noncontinuous function.

Quadratic Programming Solution

At each major iteration of the SQP method a QP problem is solved of the form where A_i refers to the `i`th row of the `m`-by-`n` matrix A.

QP

$$\begin{aligned} \underset{d \in \Re^n}{\text{minimize}} \quad & q(d) = \frac{1}{2} d^T H d + c^T d \\ & A_i d = b_i \qquad i = 1, \ldots, m_e \\ & A_i d \le b_i \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{2-30}$$

The method used in the Optimization Toolbox is an active set strategy (also known as a projection method) similar to that of Gill et al., described in [16] and [17]. It has been modified for both LP and QP problems.

The solution procedure involves two phases: the first phase involves the calculation of a feasible point (if one exists), the second phase involves the generation of an iterative sequence of feasible points that converge to the solution. In this method an active set is maintained, A_k, which is an estimate of the active constraints (i.e., which are on the constraint boundaries) at the solution point. Virtually all QP algorithms are active set methods. This point is emphasized because there exist many different methods that are very similar in structure but that are described in widely different terms.

$\bar{A}_k$ is updated at each iteration, k, and this is used to form a basis for a search direction $\hat{d}_k$. Equality constraints always remain in the active set, $\bar{A}_k$. The notation for the variable, $\hat{d}_k$, is used here to distinguish it from d_k in the major iterations of the SQP method. The search direction, $\hat{d}_k$, is calculated and minimizes the objective function while remaining on any active constraint boundaries. The feasible subspace for $\hat{d}_k$ is formed from a basis, Z_k whose columns are orthogonal to the estimate of the active set $\bar{A}_k$ (i.e., $\bar{A}_k Z_k = 0$). Thus a search direction, which is formed from a linear summation of any combination of the columns of Z_k, is guaranteed to remain on the boundaries of the active constraints.

The matrix Z_k is formed from the last m-l columns of the QR decomposition of the matrix $\bar{A}_k$, where l is the number of active constraints and $l < m$. That is, Z_k is given by

$$Z_k = Q[:, l+1:m]$$

$$\text{where} \qquad Q^T \bar{A}_k^T = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{2-31}$$

Having found Z_k, a new search direction $\hat{d}_k$ is sought that minimizes $q(d)$ where $\hat{d}_k$ is in the null space of the active constraints, that is, $\hat{d}_k$ is a linear combination of the columns of Z_k: $\hat{d}_k = Z_k p$ for some vector p.

Then if we view our quadratic as a function of p, by substituting for $\hat{d}_k$, we have

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p \tag{2-32}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c \tag{2-33}$$

$\nabla q(p)$ is referred to as the projected gradient of the quadratic function because it is the gradient projected in the subspace defined by Z_k. The term $Z_k^T H Z_k$ is called the projected Hessian. Assuming the Hessian matrix H is positive definite (which is the case in this implementation of SQP), then the minimum of the function *q(p)* in the subspace defined by Z_k occurs when $\nabla q(p) = 0$, which is the solution of the system of linear equations

$$Z_k^T H Z_k p = -Z_k^T c \tag{2-34}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k \qquad \text{where } \hat{d}_k = Z_k^T p \tag{2-35}$$

At each iteration, because of the quadratic nature of the objective function, there are only two choices of step length α. A step of unity along $\hat{d}_k$ is the exact step to the minimum of the function restricted to the null space of $\hat{A}_k$. If such a step can be taken, without violation of the constraints, then this is the solution to QP (Eq. 2.31). Otherwise, the step along $\hat{d}_k$ to the nearest constraint is less than unity and a new constraint is included in the active set at the next iterate. The distance to the constraint boundaries in any direction $\hat{d}_k$ is given by

$$\alpha = \min_i \left\{ \frac{-(A_i x_k - b_i)}{A_i \hat{d}_k} \right\} \qquad (i = 1, \ldots, m) \tag{2-36}$$

which is defined for constraints not in the active set, and where the direction $\hat{d}_k$ is towards the constraint boundary, i.e., $A_i \hat{d}_k > 0,\ i = 1, \ldots, m$.

When n independent constraints are included in the active set, without location of the minimum, Lagrange multipliers, λ_k are calculated that satisfy the nonsingular set of linear equations

$$\bar{A}_k^{T} \lambda_k = c \tag{2-37}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Eq. 2.31). However, if any component of λ_k is negative, and it does not correspond to an equality constraint, then the corresponding element is deleted from the active set and a new iterate is sought.

Initialization

The algorithm requires a feasible point to start. If the current point from the SQP method is not feasible, then a point can be found by solving the linear programming problem

$$\underset{\gamma \in \Re,\, x \in \Re^n}{\text{minimize}} \quad \gamma$$

$$A_i x = b_i \qquad i = 1, \ldots, m_e \tag{2-38}$$

$$A_i x - \gamma \le b_i \qquad i = m_e + 1, \ldots, m$$

The notation A_i indicates the ith row of the matrix A. A feasible point (if one exists) to Eq. 2.38 can be found by setting x to a value that satisfies the equality constraints. This can be achieved by solving an under- or over-determined set of linear equations formed from the set of equality constraints. If there is a solution to this problem, then the slack variable γ is set to the maximum inequality constraint at this point.

The above QP algorithm is modified for LP problems by setting the search direction to the steepest descent direction at each iteration where g_k is the gradient of the objective function (equal to the coefficients of the linear objective function)

$$\hat{d}_k = -Z_k Z_k^T g_k \tag{2-39}$$

If a feasible point is found using the above LP method, the main QP phase is entered. The search direction $\hat{d}_k$ is initialized with a search direction $\hat{d}_1$ found from solving the set of linear equations

$$H\hat{d}_1 = -g_k \tag{2-40}$$

where g_k is the gradient of the objective function at the current iterate x_k (i.e., $Hx_k + c$).

If a feasible solution is not found for the QP problem, the direction of search for the main SQP routine $\hat{d}_k$ is taken as one that minimizes γ.

Line Search and Merit Function

The solution to the QP sub-problem produces a vector d_k , which is used to form a new iterate

$$x_{k+1} = x_k + \alpha d_k \tag{2-41}$$

The step length parameter α_k is determined in order to produce a sufficient decrease in a merit function. The merit function used by Han [15] and Powell [15] of the form below has been used in this implementation

Merit Function

$$\Psi(x) = f(x) + \sum_{i=1}^{m_e} r_i \cdot g_i(x) + \sum_{i=m_e+1}^{m} r_i \cdot \max\{0, g_i(x)\} \qquad \textbf{(2-42)}$$

Powell recommends setting the penalty parameter

$$r_i = (r_{k+1})_i = \max_i\left\{\lambda_i, \frac{1}{2}((r_k)_i + \lambda_i)\right\}, \qquad i = 1, \ldots, m \qquad \textbf{(2-43)}$$

This allows positive contribution form constraints that are inactive in the QP solution but were recently active. In this implementation, initially the penalty parameter r_i is set to

$$r_i = \frac{\| \nabla f(x) \|}{\| \nabla g_i(x) \|} \qquad \textbf{(2-44)}$$

where $\| \cdot \|$ represents the Euclidean norm.

This ensures larger contributions to the penalty parameter from constraints with smaller gradients, which would be the case for active constraints at the solution point.

Multiobjective Optimization

The rigidity of the mathematical problem posed by the general optimization formulation given in GP (Eq. 2-1) is often remote from that of a practical design problem. Rarely does a single objective with several hard constraints adequately represent the problem being faced. More often there is a vector of objectives $F(x) = \{F_1(x), F_2(x), \ldots, F_m(x)\}$ 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 trade-offs between the objectives fully understood. As the number of objectives increases, trade-offs are likely to become complex and less easily quantified. There is much reliance on the intuition of the designer and his or her ability to express preferences throughout the optimization cycle. Thus, requirements for a multiobjective design strategy are to enable a natural problem formulation to be expressed, yet be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

This section begins with an introduction to multiobjective optimization, looking at a number of alternative methods. Attention is focused on the Goal Attainment method, which can be posed as a nonlinear programing problem. Algorithm improvements to the SQP method are presented for use with the Goal Attainment method.

Introduction

Multiobjective optimization is concerned with the minimization of a vector of objectives *F(x)* that may be the subject of a number of constraints or bounds.

MO

$$\begin{array}{l} \underset{x \in \Re^n}{\text{minimize}} \quad F(x) \\ \qquad G_i(x) = 0 \qquad i = 1, \ldots, m_e \\ \qquad G_i(x) \le 0 \qquad i = m_e + 1, \ldots, m \\ \qquad x_l \le x \le x_u \end{array} \tag{2-45}$$

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 [25] (also called Pareto optimality [24], [26]) 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 $x \in \Re^n$ that satisfies all the constraints, i.e.,

$$\begin{aligned} &\Omega = \{x \in \Re^n\} \\ \text{subject to} \quad & g_i(x) = 0 \qquad i = 1, \ldots, m_e \\ & g_i(x) \le 0 \qquad i = m_e + 1, \ldots, m \\ & x_l \le x \le x_u \end{aligned} \tag{2-46}$$

This allows us to define the corresponding feasible region for the objective function space Λ

$$\Lambda = \{y \in \Re^m\} \qquad \text{where} \qquad y = F(x) \qquad \text{subject to} \qquad x \in \Omega. \tag{2-47}$$

The performance vector, *F(x)*, maps parameter space into objective function space as is represented for a two-dimensional case in Figure 2-9 below.

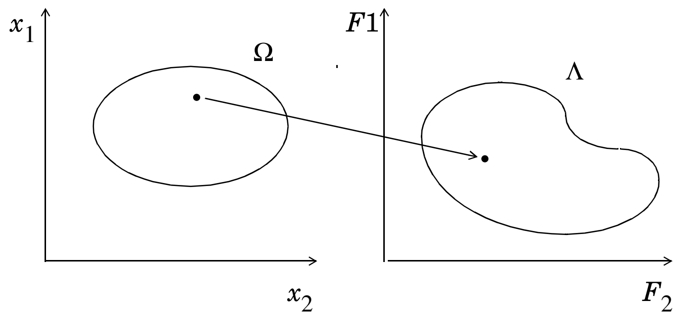


Figure 2-9: Mapping from Parameter Space into Objective Function Space

A noninferior solution point can now be defined.

Definition: A 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

$$\begin{aligned} &F_i(x^* + \Delta x) \le F_i(x^*) \qquad i = 1, \ldots, m \\ &F_j(x^* + \Delta x) < F_j(x^*) \qquad \text{for some } j. \end{aligned} \tag{2-48}$$

In the two-dimensional representation of Figure 2-10 the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.

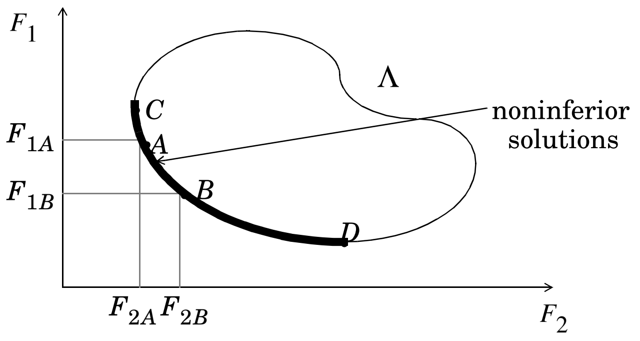


Figure 2-10: Set of Noninferior Solutions

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 not a noninferior 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. The techniques for multiobjective optimization are wide and varied and all the methods cannot be covered within the scope of this toolbox. However, some of the techniques are described below.

Weighted Sum Strategy

The weighted sum strategy converts the multiobjective problem of minimizing the vector $F(x)$ into a scalar problem by constructing a weighted sum of all the objectives.

Weighted Sum

$$\underset{x \in \Omega}{\text{minimize}} \quad f(x) = \sum_{i=1}^{m} w_i \cdot F_i(x)^2 \tag{2-49}$$

The problem can then be optimized using a standard unconstrained optimization algorithm. The problem here is in attaching weighting coefficients to each of the objectives. The weighting coefficients do not necessarily correspond directly to the relative importance of the objectives or allow trade-offs between the objectives to be expressed. Further, the noninferior solution boundary may be nonconcurrent so that certain solutions are not accessible.

This can be illustrated geometrically. Consider the two objective case in Figure . In the objective function space a line, L, $w^T F(x) = c$ is drawn. The minimization of Eq. 2-49 can be interpreted as finding the value of c for which L just touches the boundary of Λ as it proceeds outwards from the origin. Selection of weights w_1 and w_2, therefore, defines the slope of L, which in turn leads to the solution point where L touches the boundary of Λ.

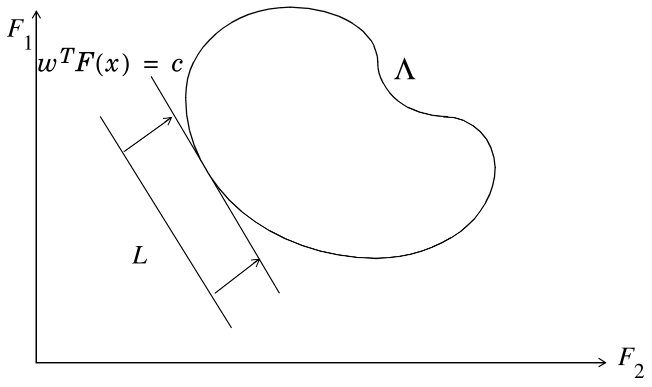


Figure 2-11: Geometrical Representation of the Weighted Sum Method

The aforementioned convexity problem arises when the lower boundary of Λ is nonconvex as shown in Figure 2-12. In this case the set of noninferior solutions between A and B is not available.

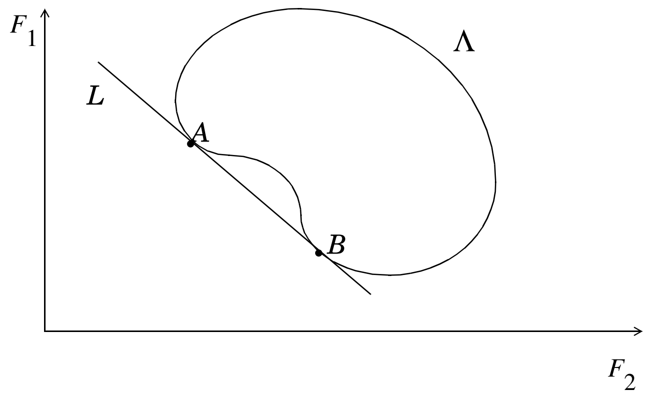


Figure 2-12: Nonconvex Solution Boundary

ε-Constraint Method

A procedure that overcomes some of the convexity problems of the weighted sum technique is the ε-constraint method. This involves minimizing a primary objective, F_p, and expressing the other objectives in the form of inequality constraints

$$\begin{aligned} &\underset{x \in \Omega}{\text{minimize}} \quad F_p(x) \\ \text{subject to} \quad & F_i(x) \le \varepsilon_i \qquad i = 1, \ldots, m \qquad i \neq p \end{aligned} \tag{2-50}$$

Figure 2-13 shows a two-dimensional representation of the ε-constraint method for a two objective problem.

$$\underset{x \in \Omega}{\text{minimize}} \; F_1(x) \qquad \text{subject to: } F_2x \le \varepsilon_2$$

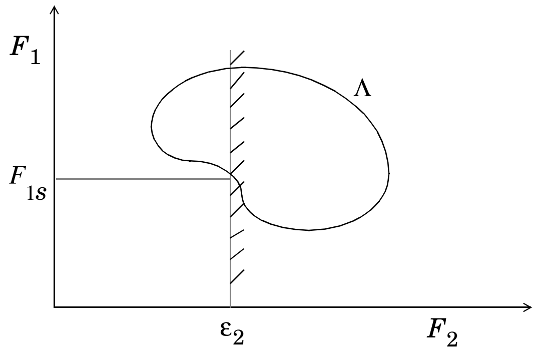


Figure 2-13: Geometrical Representation of ε-Constraint Method

This approach is able to identify a number of noninferior solutions on a nonconvex boundary that are not obtainable using the weighted sum technique, for example, at the solution point $F_1 = F_{1s}$ and $F_2 = \varepsilon_2$. A problem with this method is, however, a suitable selection of ε to ensure a feasible solution. A further disadvantage of this approach is that the use of hard constraints is rarely adequate for expressing true design objectives. Similar methods exist, such as that of Waltz [31], which prioritize the objectives. The optimization proceeds with reference to these priorities and allowable bounds of acceptance. The difficulty here is in expressing such information at early stages of the optimization cycle.

In order for the designers' true preferences to be put into a mathematical description, the designers must express a full table of their preferences and satisfaction levels for a range of objective value combinations. A procedure must then be realized that is able to find a solution with reference to this. Such methods have been derived for discrete functions using the branches of statistics known as decision theory and game theory (for a basic introduction, see [28]). Implementation for continuous functions requires suitable discretization strategies and complex solution methods. Since it is rare for the designer to know such detailed information, this method is deemed impractical for most practical design problems. It is, however, seen as a possible area for further research.

What is required is a formulation that is simple to express, retains the designers preferences, and is numerically tractable.

Goal Attainment Method

The method described here is the Goal Attainment method of Gembicki [27]. This involves expressing a set of design goals, $F^* = \{F_1^*, F_2^*, \ldots, F_m^*\}$, which is associated with a set of objectives, $F(x) = \{F_1(x), F_2(x), \ldots, F_m(x)\}$. The problem formulation allows the objectives to be under- or over-achieved enabling the designer to be relatively imprecise about initial design goals. The relative degree of under- or over-achievement of the goals is controlled by a vector of weighting coefficients, $w = \{w_1, w_2, \ldots, w_m\}$, and is expressed as a standard optimization problem using the following formulation:

Goal Attainment

$$\begin{array}{lll} & \underset{\gamma \in \Re,\, x \in \Omega}{\text{minimize}} \quad \gamma & \\ \text{such that} & F_i(x) - w_i\gamma \le F_i^* & i = 1, \ldots, m \end{array} \qquad \textbf{(2-51)}$$

The term $w_i\gamma$ introduces an element of *slackness* into the problem, which otherwise imposes that the goals be rigidly met. The weighting vector, w, enables the designer to express a measure of the relative trade-offs between the objectives. For instance, setting the weighting vector, w, equal to the initial goals indicates that the same percentage under- or over-attainment of the goals, F^*, is achieved. Hard constraints can be incorporated into the design by setting a particular weighting factor to zero (i.e., $w_i = 0$). The Goal Attainment method provides a convenient intuitive interpretation of the design problem, which is solvable using standard optimization procedures. Illustrative examples of the use of Goal Attainment method in control system design can be found in Fleming [29,30].

The Goal Attainment method is represented geometrically in Figure 2-14 for the two-dimensional problem.

$$\underset{\gamma,\, x \in \Omega}{\text{minimize}}\ \gamma \quad \text{subject to:} \quad \begin{aligned} F_1(x) - w_1\gamma &\le F_1^* \\ F_2(x) - w_2\gamma &\le F_2^* \end{aligned}$$

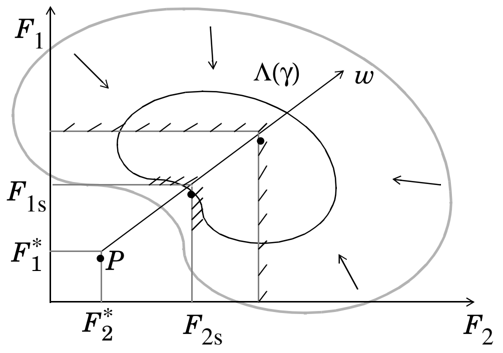


Figure 2-14: Geometrical Representation of Goal Attainment Method

Specification of the goals, $\{F_1^*, F_2^*\}$, defines the goal point, P. The weighting vector defines the direction of search from P to the feasible function space, $\Lambda(\gamma)$. During the optimization γ is varied, which changes the size of the feasible region. The constraint boundaries converge to the unique solution point F_{1s}, F_{2s}.

Algorithm Improvements for Goal Attainment Method

The Goal Attainment method has the advantage that it can be posed as a nonlinear programming problem. Characteristics of the problem can also be exploited in a nonlinear programming algorithm. In Sequential Quadratic Programming (SQP) the choice of merit function for the line search is not easy because, in many cases, it is difficult to “define” the relative importance between improving the objective function and reducing constraint violations. This has resulted in a number of different schemes for constructing the merit function (see, for example, Schittowski [22]). In Goal Attainment programming there may be a more appropriate merit function, which can be achieved by posing Eq. 2-51 as the minimax problem

$$\underset{x \in \Re^n}{\text{minimize}} \ \max_i \ \{\Lambda_i\}$$

$$\text{where} \qquad \Lambda_i = \frac{F_i(x) - F_i^*}{w_i} \qquad i = 1, \ldots, m \tag{2-52}$$

Following the argument of Brayton et al. [32] for minimax optimization using SQP, using the merit function of Eq. 2-43 for the Goal Attainment problem of Eq. 2-52, gives

$$\psi(x, \gamma) = \gamma + \sum_{i=1}^{m} r_i \cdot \max\ \{0, F_i(x) - w_i\gamma - F_i^*\} \tag{2-53}$$

When the merit function of Eq. 2-53 is used as the basis of a line search procedure, then, although $\psi(x, \gamma)$ may decrease for a step in a given search direction, the function `max` Λ_i may paradoxically increase. This is accepting a degradation in the worst case objective. Since the worst case objective is responsible for the value of the objective function γ, this is accepting a step that ultimately increases the objective function to be minimized. Conversely, $\psi(x, \gamma)$ may increase when `max` Λ_i decreases implying a rejection of a step that improves the worst case objective.

Following the lines of Brayton et al. [32], a solution is therefore to set $\psi(x)$ equal to the worst case objective, i.e.,

$$\psi(x) = \max_i \ \Lambda_i. \tag{2-54}$$

A problem in the Goal Attainment method is that it is common to use a weighting coefficient equal to zero to incorporate hard constraints. The merit function of Eq. 2-54 then becomes infinite for arbitrary violations of the constraints. To overcome this problem while still retaining the features of Eq. 2-54 the merit function is combined with that of Eq. 2-43 giving the following:

$$\psi(x) = \sum_{i=1}^{m} \begin{cases} r_i \cdot \max\ \{0, F_i(x) - w_i\gamma - F_i^*\} & \text{if } w_i = 0 \\ \max_i \ \Lambda_i, \quad i = 1, \ldots, m & \text{otherwise} \end{cases} \tag{2-55}$$

Another feature that can be exploited in SQP is the objective function γ. From the KT equations (Eq. 2-24) it can be shown that the approximation to the

Hessian of the Lagrangian, H**,** should have zeros in the rows and columns associated with the variable γ. By initializing H as the identity matrix, this property does not appear. H is therefore initialized and maintained to have zeros in the rows and columns associated with γ.

These changes make the Hessian, H, indefinite, therefore H is set to have zeros in the rows and columns associated with γ, except for the diagonal element, which is set to a small positive number (e.g., `1e–10`). This allows use of the fast converging positive definite QP method described in the “Quadratic Programming Solution” section.

The above modifications have been implemented in `fgoalattain` and have been found to make the method more robust. However, due to the rapid convergence of the SQP method, the requirement that the merit function strictly decrease sometimes requires more function evaluations than an implementation of SQP using the merit function of (Eq. 2-43).

Review

A number of different optimization strategies have been discussed. The algorithms used (e.g., BFGS, Levenberg-Marquardt and SQP) have been chosen for their robustness and iterative efficiency. The choice of problem formulation (e.g., unconstrained, least squares, constrained, minimax, multiobjective, or goal attainment) depends on the problem being considered and the required execution efficiency.

References

[1] Gill, P.E., W. Murray, and M.H.Wright, *Practical Optimization*, Academic Press, London, 1981.

[2] Fletcher, R., “Practical Methods of Optimization,” *Vol. 1, Unconstrained Optimization*, and *Vol. 2, Constrained Optimization,* John Wiley and Sons., 1980.

[3] Broyden, C.G., “The Convergence of a Class of Double-rank Minimization Algorithms,” *J.Inst. Maths. Applics*., Vol. 6, pp. 76-90, 1970.

[4] Fletcher, R., “A New Approach to Variable Metric Algorithms,” *Computer Journal*, Vol. 13, pp. 317-322, 1970.

[5] Goldfarb, D., “A Family of Variable Metric Updates Derived by Variational Means,” *Mathematics of Computing*, Vol. 24, pp. 23-26, 1970.

[6] Shanno, D.F., “Conditioning of Quasi-Newton Methods for Function Minimization,” *Mathematics of Computing*, Vol. 24, pp. 647-656, 1970.

[7] Davidon, W.C., “Variable Metric Method for Minimization,” *A.E.C. Research and Development Report*, ANL-5990, 1959.

[8] Fletcher, R. and M.J.D. Powell, “A Rapidly Convergent Descent Method for Minimization,” *Computer J.*, Vol. 6, pp. 163-168, 1963.

[9] Biggs, M.C., “Constrained Minimization Using Recursive Quadratic Programming,” *Towards Global Optimization* (L.C.W.Dixon and G.P.Szergo, eds.), North-Holland, pp. 341-349, 1975.

[10] Han, S.P., “A Globally Convergent Method for Nonlinear Programming,” *J. Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

[11] Powell, M.J.D., “A Fast Algorithm for Nonlinearly Constrained Optimization Calculations,” *Numerical Analysis*, G.A.Watson ed., Lecture Notes in Mathematics, Springer Verlag, Vol. 630, 1978.

[12] Powell, M.J.D., “The Convergence of Variable Metric Methods for Nonlinearly Constrained Optimization Calculations,” *Nonlinear Programming 3*, (O.L. Mangasarian, R.R. Meyer and S.M. Robinson, eds.), Academic Press, 1978.

[13] Powell, M.J.D., “Variable Metric Methods for Constrained Optimization,” *Mathematical Programming: The State of the Art*, (A.Bachem, M.Grotschel and B.Korte, eds.) Springer Verlag, pp. 288-311, 1983.

[14] Hock, W. and K. Schittowski, “A Comparative Performance Evaluation of 27 Nonlinear Programming Codes,” *Computing*, Vol. 30, p. 335, 1983.

[15] Dantzig, G., *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[16] Gill, P.E., W. Murray, and M.H. Wright, *Numerical Linear Algebra and Optimization*, Vol. 1, Addison Wesley, 1991.

[17] Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright, “Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints,” *ACM Trans. Math. Software*, Vol. 10, pp. 282-298, 1984.

[18] Levenberg, K., “A Method for the Solution of Certain Problems in Last Squares,” *Quart. Appl. Math*. Vol. 2, pp. 164-168, 1944.

[19] Marquardt, D., “An Algorithm for Least-Squares Estimation of Nonlinear Parameters,” *SIAM J. Appl. Math*. Vol. 11, pp. 431-441, 1963.

[20] Moré, J.J., “The Levenberg-Marquardt Algorithm: Implementation and Theory,” *Numerical Analysis*, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp. 105-116, 1977.

[21] Dennis, J.E., Jr., “Nonlinear Least Squares,” *State of the Art in Numerical Analysis* ed. D. Jacobs, Academic Press, pp. 269-312, 1977.

[22] Schittowski, K., “NLQPL: A FORTRAN-Subroutine Solving Constrained Nonlinear Programming Problems,” *Annals of Operations Research*, Vol. 5, pp. 485-500, 1985.

[23] *NAG Fortran Library Manual,* Mark 12, Vol. 4, E04UAF, p. 16.

[24] Censor, Y., “Pareto Optimality in Multiobjective Problems, ” *Appl. Math. Optimiz*., Vol. 4, pp. 41-59, 1977.

[25] Zadeh, L.A., “Optimality and Nonscalar-valued Performance Criteria,” *IEEE Trans. Automat. Contr*., Vol. AC-8, p. 1, 1963.

[26] 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.

[27] Gembicki, F.W., “Vector Optimization for Control with Performance and Parameter Sensitivity Indices,” Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, Ohio, 1974.

[28] Hollingdale, S.H., *Methods of Operational Analysis in Newer Uses of Mathematics* (James Lighthill, ed.), Penguin Books, 1978.

[29] Fleming, P.J., “Application of Multiobjective Optimization to Compensator Design for SISO Control Systems,” *Electronics Letters*, Vol. 22, No. 5, pp. 258-259, 1986.

[30] Fleming, P.J., “Computer-Aided Control System Design of Regulators using a Multiobjective Optimization Approach,” *Proc. IFAC Control Applications of Nonlinear Porg. and Optim.*, Capri, Italy, pp. 47-52, 1985.

[31] Waltz, F.M., “An Engineering Approach: Hierarchical Optimization Criteria,” *IEEE Trans.*, Vol. AC-12, pp. 179-180, April, 1967.

[32] Brayton, R.K., S.W. Director, G.D. Hachtel, and L.Vidigal, “A New Algorithm for Statistical Circuit Design Based on Quasi-Newton Methods and Function Splitting,” *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, pp. 784-794, Sept. 1979.

[33] Nelder, J.A. and R. Mead, “A Simplex Method for Function Minimization,” *Computer J.,* Vol .7, pp. 308-313, 1965.

[34] Grace, A.C.W., “Computer–Aided Control System Design Using Optimization Techniques,” Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

[35] Madsen, K. and H. Schjaer-Jacobsen, *“Algorithms for Worst Case Tolerance Optimization,” IEEE Transactions of Circuits and Systems*, Vol. CAS-26, Sept. 1979.

[36] Forsythe, G.F., M.A. Malcolm, and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice Hall, 1976.

[37] Dantzig, G.B., A. Orden, and P. Wolfe, “Generalized Simplex Method for Minimizing a Linear from Under Linear Inequality Constraints,” *Pacific J. Math*. Vol. 5, pp. 183-195.

Large-Scale Algorithms

Large-Scale Optimization

To solve large-scale optimization problems, special techniques are needed. This chapter describes the techniques used in the Optimization Toolbox for the large-scale methods, in particular trust region methods, preconditioned conjugate gradients, projection methods for equality constraints, and reflective Newton methods for bound constraints. Finally, we describe interior-point techniques for linear programming, specifically a primal-dual Mehrotra predictor-corrector method.

Trust Region Methods for Nonlinear Minimization

Many of the methods used in the Optimization Toolbox are based on the trust region idea, a simple yet powerful concept in optimization.

To understand the trust region approach to optimization, consider the unconstrained minimization problem, minimize $f(x)$, where the function takes vector arguments and returns scalars. Suppose we are at a point x in n-space and we want to improve, i.e., move to a point with a lower function value. The basic idea is to approximate f with a simpler function q, which reasonably reflects the behavior of function f in a neighborhood N around the point x. This neighborhood is the *trust region*. A trial step s is computed by minimizing (or approximately minimizing) over N. This is the trust region subproblem,

$$\min_{s} \{ q(s) \text{such that } s \in N \} \tag{3-1}$$

The current point is updated to be $x + s$ if $f(x + s) < f(x)$; otherwise, the current point remains unchanged and N, the region of trust, is shrunk and the trial step computation is repeated.

The key questions in defining a specific trust region approach to minimizing $f(x)$ are how to choose and compute the approximation q (defined at the current point x), how to choose and modify the trust region N, and how accurately to solve the trust region subproblem. In this section we focus on the unconstrained problem; additional complications due to the presence of constraints on the variables are discussed in later sections.

In the standard trust region method ([2]), the quadratic approximation q is defined by the first two terms of the Taylor approximation to f at x; the neighborhood N is usually spherical or ellipsoidal in shape. Mathematically the trust region subproblem is typically stated

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \ \text{ such that } \ \|Ds\| \le \Delta \right\} \tag{3-2}$$

where g is the gradient of f at the current point x, H is the Hessian matrix (the symmetric matrix of second derivatives), D is a diagonal scaling matrix, Δ is a positive scalar, and $\|\cdot\|$ is the 2-norm. Good algorithms exist for solving Eq. 3-2 (see [2]); such algorithms typically involve the computation of a full eigensystem and a Newton process applied to the secular equation

$$\frac{1}{\Delta} - \frac{1}{\|s\|} = 0$$

Such algorithms provide an accurate solution to Eq. 3-2, however they require time proportional to several factorizations of H; therefore, for large-scale problems a different approach is needed. Several approximation and heuristic strategies, based on Eq. 3-2, have been proposed in the literature ([2],[10]). The approximation approach we follow in the Optimization Toolbox is to restrict the trust region subproblem to a two-dimensional subspace S ([2],[1]). Once the subspace S has been computed, the work to solve Eq. 3-2 is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The toolbox assigns $S = \langle s_1, s_2 \rangle$ where s_1 is in the direction of the gradient g, and s_2 is either an approximate Newton direction, i.e., a solution to

$$H \cdot s_2 = -g \qquad \textbf{(3-3)}$$

or a direction of negative curvature

$$s_2^T \cdot H \cdot s_2 < 0 \qquad \textbf{(3-4)}$$

The philosophy behind this choice of S is to force global convergence (via the steepest descent direction or negative curvature direction) and achieve fast local convergence (via the Newton step, when it exists).

A framework for the Optimization Toolbox approach to unconstrained minimization using trust region ideas is now easy to describe:

- Formulate the two-dimensional trust region subproblem
- Solve Eq. 3-2 to determine the trial step s
- If $f((x+s) \le f(x))$ then $x = x + s$
- Adjust Δ

These four steps are repeated until convergence. The trust region dimension Δ is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e., $f(x+s) \ge f(x)$. See [9],[6] for a discussion of this aspect.

The Optimization Toolbox treats a few important special cases of *f* with specialized functions: nonlinear least-squares, quadratic functions, and linear least-squares. However, the underlying algorithmic ideas are the same as for the general case. These special cases are discussed in later sections.

Preconditioned Conjugate Gradients

A popular way to solve large symmetric positive definite systems of linear equations $Hp = -g$ is the method of preconditioned conjugate gradients (PCG). This iterative approach requires the ability to perform matrix-vector products of the form $H \cdot v$ where v is an arbitrary vector. The matrix positive definite matrix M is a *preconditioner* for H, that is, $M = C^2$ where $C^{-1}HC^{-1}$ is well-conditioned matrix or a matrix with clustered eigenvalues.

Algorithm PCG

```
% Initializations
r = -g; p = zeros(n,1);
% Precondition
z = M\r; inner1 = r'*z; inner2 = 0; d = z;
% Conjugate gradient iteration
for k = 1:kmax
   if k > 1
       beta = inner1/inner2;
       d = z + beta*d;
   end
w = H*d; denom = d'*w;
   if denom <= 0
       p = d/norm(d); % Direction of negative/zero curvature
       break % Exit if zero/negative curvature detected
   else
       alpha = inner1/denom;
       p = p + alpha*d;
       r = r - alpha*w;
   end
   z = M\r;
   if norm(z) <= tol % Exit if Hp=-g solved within tolerance
       break
   end
   inner2 = inner1;
   inner1 = r'*z;
end
```

In a minimization context you can assume that the Hessian matrix H is symmetric. However, H is guaranteed to be positive definite only in the neighborhood of a strong minimizer. Algorithm PCG exits when a direction of negative (or zero) curvature is encountered, i.e., $d^T H d \le 0$. The PCG output direction, p, is either a direction of negative curvature or an approximate (*tol* controls how approximate) solution to the Newton system $Hp = -g$. In either case p is used to help define the two-dimensional subspace used in the trust region approach discussed in the section "Trust Region Methods for Nonlinear Minimization."

Linearly Constrained Problems

Linear constraints complicate the situation described for unconstrained minimization; however, the underlying ideas described above can be carried through in a clean and efficient way. The large-scale methods in the Optimization Toolbox generate strictly feasible iterates: a projection technique is used for linear equality constraints, reflections are used with simple box constraints.

Linear Equality Constraints

The general linear equality constrained minimization problem can be written

$$\min \{ f(x) \text{ such that } Ax = b \} \tag{3-5}$$

where A is an m-by-n matrix ($m \le n$). The Optimization Toolbox preprocesses A to remove strict linear dependencies using a technique based on the LU-factorization of A^T ([6]). Here we will assume A is of rank m.

Our method to solve Eq. 3-5 differs from our unconstrained approach in two significant ways. First, an initial feasible point x_0 is computed, using a sparse least-squares step, so that $Ax_0 = b$. Second, Algorithm PCG is replaced with Reduced Preconditioned Conjugate Gradients (RPCG), see [6], in order to compute an approximate reduced Newton step (or a direction of negative curvature in the null space of A). The key linear algebra step involves solving systems of the form

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{3-6}$$

where $\tilde{A}$ approximates A (small nonzeros of A are set to zero provided rank is not lost) and C is a sparse symmetric positive-definite approximation to H, i.e., $C^T C \approx H$. See [6] for more details.

Box Constraints

The box constrained problem is of the form

$$\min \{ f(x) \text{ such that } l \le x \le u \} \tag{3-7}$$

where l is a vector of lower bounds, and u is a vector of upper bounds. Some (or all) of the components of l may be equal to $-\infty$ and some (or all) of the components of u may be equal to ∞. The method generates a sequence of strictly feasible points. Two techniques are used to maintain feasibility while achieving robust convergence behavior. First, a scaled modified Newton step replaces the unconstrained Newton step (to define the two-dimensional subspace S). Second, reflections are used to increase the stepsize.

The scaled modified Newton step arises from examining the Kuhn-Tucker necessary conditions for Eq. 3-7

$$(D(x))^{-2}g = 0 \qquad \textbf{(3-8)}$$

where

$$D(x) = \mathrm{diag}\left(\left|v_k\right|^{-\frac{1}{2}}\right)$$

and the vector $v(x)$ is defined below, for each $1 \le i \le n$:

- If $g_i < 0$ and $u_i < \infty$ then $v_i = x_i - u_i$
- If $g_i \ge 0$ and $l_i > -\infty$ then $v_i = x_i - l_i$
- If $g_i < 0$ and $u_i = \infty$ then $v_i = -1$
- If $g_i \ge 0$ and $l_i = -\infty$ then $v_i = 1$

The nonlinear system Eq. 3-8 is not differentiable everywhere; nondifferentiability occurs when $v_i = 0$. Hence we avoid such points by maintaining strict feasibility, i.e., restricting $l < x < u$.

The scaled modified Newton step s_k^N for Eq. 3-8 is defined as the solution to the linear system

$$\hat{M}Ds^N = -\hat{g} \qquad \textbf{(3-9)}$$

where

$$\hat{g} = D^{-1}g = \mathrm{diag}\left(|v|^{\frac{1}{2}}\right)g \qquad \textbf{(3-10)}$$

and

$$\hat{M} = D^{-1}HD^{-1} + \mathrm{diag}(g)J^{v} \tag{3-11}$$

Here J^v plays the role of the Jacobian of $|v|$. Each diagonal component of the diagonal matrix J^v equals 0, -1 or 1. If all the components of l and u are finite, $J^v = \mathrm{diag}(\mathrm{sign}(g))$. At a point where $g_i = 0$, v_i may not be differentiable. We define $J^v_{ii} = 0$ at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value v_i takes. Further $|v_i|$ will still be discontinuous at this point, but the function $|v_i| \cdot g_i$ is continuous.

Second, reflections are used to increase the stepsize. A (single) reflection step is defined as follows. Given a step p that intersects a bound constraint, consider the first bound constraint crossed by p; assume it is the ith bound constraint (either the ith upper or ith lower bound). Then the reflection step $p^R = p$ except in the ith component where $p^R_i = -p_i$.

Nonlinear Least-Squares

An important special case for *f(x)* is the nonlinear least-squares problem

$$f(x) = \frac{1}{2}\sum_i f_i^2(x) = \frac{1}{2}\|F(x)\|_2^2 \qquad \textbf{(3-12)}$$

where $F(x)$ is a vector-valued function with component i of $F(x)$ equal to $f_i(x)$. The basic method used to solve this problem is the same as in the general case described in "Trust Region Methods for Nonlinear Minimization." However, the structure of the nonlinear least-squares problem is exploited to enhance efficiency. In particular, an approximate Gauss-Newton direction, i.e., a solution s to

$$\min \quad \|Js + F\|_2^2 \qquad \textbf{(3-13)}$$

(where J is the Jacobian of $F(x)$) is used to help define the two-dimensional subspace S . Second derivatives of the component function $f_i(x)$ are not used.

In each iteration the method of preconditioned conjugate gradients is used to approximately solve the normal equations, i.e.,

$$J^T Js = -J^T F$$

although the normal equations are not explicitly formed.

Quadratic Programming

In this case the function *f(x)* is the quadratic equation

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

The subspace trust region method is used to determine a search direction. However, instead of restricting the step to (possibly) one reflection step as in the nonlinear minimization case, a piecewise reflective line search is conducted at each iteration. See [5] for details of the line search.

Linear Least-Squares

In this case the function *f(x)* to be solved is

$$f(x) = \frac{1}{2}x^T C^T C x + x^T C^T d$$

The algorithm generates strictly feasible iterates converging, in the limit, to a local solution. Each iteration involves the approximate solution of a large linear system (of order n, where n is the length of x); the iteration matrices have the structure of the matrix C. In particular, the method of preconditioned conjugate gradients is used to approximately solve the normal equations, i.e,

$$C^T C x = -C^T d$$

although the normal equations are not explicitly formed.

The subspace trust region method is used to determine a search direction. However, instead of restricting the step to (possibly) one reflection step as in the nonlinear minimization case, a piecewise reflective line search is conducted at each iteration, as in the quadratic case. See [5] for details of the line search. Ultimately, the linear systems represent a Newton approach capturing the first-order optimality conditions at the solution: resulting in strong local convergence rates.

Large-Scale Linear Programming

Linear programming is defined as

$$\min\ f^T x \quad \text{such that} \begin{pmatrix} Aeq \cdot x = beq \\ Aineq \cdot x \le bineq \\ l \le x \le u \end{pmatrix} \tag{3-14}$$

The large-scale method is based on LIPSOL ([11]), which is a variant of Mehrotra's predictor-corrector algorithm ([7]), a primal-dual interior-point method.

Main Algorithm

The algorithm begins by applying a series of preprocessing steps (see "Preprocessing" on page 3-17). After preprocessing, the problem has the form

$$\min\ f^T x \qquad \text{such that} \begin{pmatrix} A \cdot x = b \\ 0 \le x \le u \end{pmatrix} \tag{3-15}$$

The upper bounds constraints are implicitly included in the constraint matrix *A* with the addition of primal slack variables *s,* Eq. 3-15 becomes

$$\min\ f^T x \qquad \text{such that} \begin{pmatrix} A \cdot x = b \\ x + s = u \\ x \ge 0, s \ge 0 \end{pmatrix} \tag{3-16}$$

which is referred to as the *primal* problem: *x* consists of the primal variables and *s* consists of the primal slack variables. The *dual* problem is

$$\max\ b^T y - u^T w \qquad \text{such that} \qquad \begin{matrix} A^T \cdot y - w + z = f \\ z \ge 0, w \ge 0 \end{matrix} \tag{3-17}$$

where *y* and *w* consist of the dual variables and *z* consists of the dual slacks. The optimality conditions for this linear program, i.e., the primal Eq. 3-16 and the dual Eq. 3-17, are

$$F(x, y, z, s, w) = \begin{pmatrix} A \cdot x - b \\ x + s - u \\ A^T \cdot y - w + z - f \\ x_i z_i \\ s_i w_i \end{pmatrix} = 0 \tag{3-18}$$

$$x \geq 0, z \geq 0, s \geq 0, w \geq 0$$

where $x_i z_i$ and $s_i w_i$ denote component-wise multiplication.

The quadratic equations $x_i z_i = 0$ and $s_i w_i = 0$ are called the *complementarity* conditions for the linear program; the other (linear) equations are called the *feasibility* conditions. The quantity

$$x^T z + s^T w$$

is the *duality gap*, which measures the residual of the complementarity portion of F when $(x, z, s, w) \geq 0$.

The algorithm is a *primal-dual algorithm* meaning that both the primal and the dual programs are solved simultaneously. It can be considered a Newton-like method, applied to the linear-quadratic system $F(x, y, z, s, w) = 0$ in Eq. 3-18, while at the same time keeping the iterates x, z, w, and s positive, thus the name interior-point method. (The iterates are in the strictly interior region represented by the inequality constraints in Eq. 3-16.)

The algorithm is a variant of the predictor-corrector algorithm proposed by Mehrotra. Consider an iterate $v = [x;\ y;\ z;\ s;\ w]$, where $[x;\ z;\ s;\ w] > 0$. We first compute the so-called *prediction* direction

$$\Delta v_p = -(F^T(v))^{-1} F(v)$$

which is the Newton direction; then the so-called *corrector* direction

$$\Delta v_c = -(F^T(v))^{-1} (F(v + \Delta v_p)) - \mu \hat{e}$$

where $\mu > 0$ is called the *centering* parameter and must be chosen carefully. $\hat{e}$ is a zero-one vector with the ones corresponding to the quadratic equations in *F(v),* i.e., the perturbations are only applied to the complementarity conditions, which are all quadratic, but not to the feasibility conditions, which are all linear. We combine the two directions with a step-length parameter $\alpha > 0$ and update v to obtain the new iterate v^+

$$v^+ = v + \alpha(\Delta v_p + \Delta v_c)$$

where the step-length parameter α is chosen so that

$$v^+ = [x^+;\ y^+;\ z^+;\ s^+;\ w^+]$$

satisfies

$$[x^+;\ z^+;\ s^+;\ w^+] > 0$$

In solving for the steps above, the algorithm computes a (sparse) direct factorization on a modification of the Cholesky factors of $A \cdot A^T$. If A has dense columns, it instead uses the Sherman-Morrison formula, and if that solution is not adequate (the residual is too large), it uses preconditioned conjugate gradients to find a solution.

The algorithm then repeats these steps until the iterates converge. The main stopping criteria is a standard one

$$\frac{\|r_b\|}{\max(1, \|b\|)} + \frac{\|r_f\|}{\max(1, \|f\|)} + \frac{\|r_u\|}{\max(1, \|u\|)} + \frac{|f^T x - b^T y + u^T w|}{\max(1, |f^T x|, |b^T y - u^T w|)} \le tol$$

where

$$r_b = Ax - b$$
$$r_f = A^T y - w + z - f$$
$$r_u = x + s - u$$

are the primal residual, dual residual, and upper-bound feasibility respectively, and

$$f^T x - b^T y + u^T w$$

is the difference between the primal and dual objective values, and *tol* is some tolerance. The sum in the stopping criteria measures the total relative errors in the optimality conditions in Eq. 3-18.

Preprocessing

A number of preprocessing steps occur before the actual iterative algorithm begins. The resulting transformed problem is one where:

- All variables are bounded below by zero.
- All constraints are equalities.
- Fixed variables, those with equal upper and lower bounds, are removed.
- Rows of all zeros in the constraint matrix are removed.
- The constraint matrix has full structural rank.
- Columns of all zeros in the constraint matrix are removed.
- When a significant number of singleton rows exist in the constraint matrix, the associated variables are solved for and the rows removed.

While these preprocessing steps can do much to speed up the iterative part of the algorithm, if the Lagrange multipliers are required, the preprocessing steps must be "undone" since the multipliers calculated during the algorithm are for the transformed, and not the original, problem. Thus, if the multipliers are *not* requested, this transformation back will not be computed, and may save some time computationally.

References

[1] Branch, M.A., T.F. Coleman, Y. Li, "A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems," to appear in a future *SIAM Journal on Scientific Computing*.

[2] Byrd, R.H., R.B. Schnabel, and G.A. Shultz, "Approximate Solution of the Trust Region Problem by Minimization over Two-Dimensional Subspaces," *Mathematical Programming*, Vol. 40, pp. 247-263, 1988.

[3] Coleman, T.F. and Y. Li, "On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds," *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.

[4] Coleman, T.F. and Y. Li, "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds," *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

[5] Coleman, T.F. and Y. Li, "A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on some of the Variables," *SIAM Journal on Optimization*, Vol. 6, Number 4, pp. 1040-1058, 1996.

[6] Coleman, T.F. and A. Verma, "On Preconditioning Conjugate Gradients for Linear Equality Constrained Minimization," submitted to *SIAM Journal on Scientific Computing*.

[7] Mehrotra, S., "On the Implementation of a Primal-Dual Interior Point Method," *SIAM Journal on Optimization*, Vol. 2, pp. 575-601, 1992.

[8] Moré, J.J. and D.C. Sorensen, "Computing a Trust Region Step," *SIAM Journal on Scientific and Statistical Computing*, Vol. 3, pp. 553-572, 1983.

[9] Sorensen, D.C., "Minimization of a Large Scale Quadratic Function Subject to an Ellipsoidal Constraint," *Department of Computational and Applied Mathematics*, Rice University, Technical Report TR94-27, 1994.

[10] Steihaug, T., "The Conjugate Gradient Method and Trust Regions in Large Scale Optimization," *SIAM Journal on Numerical Analysis*, Vol. 20, pp. 626-637, 1983.

[11] Zhang, Y., "Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment," Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, MD, Technical Report TR96-01, July, 1995.

Reference

This chapter contains descriptions of the Optimization Toolbox functions, listed alphabetically. The chapter starts with tables listing the types of optimization and which functions apply. The next set of tables lists general descriptions of all the input and output arguments and the parameters in the optimization options structure, and which functions use those arguments or parameters. Information specific to a function about input arguments, output arguments, and options is listed within that function's reference pages under the *Arguments* section, after the function description.

Information is also available through the online Help facility.

Optimization Functions

Minimization

Function	Purpose
`fgoalattain`	Multiobjective goal attainment
`fminbnd`	Scalar nonlinear minimization with bounds
`fmincon`	Constrained nonlinear minimization
`fminimax`	Minimax optimization
`fminsearch`, `fminunc`	Unconstrained nonlinear minimization
`fseminf`	Semi-infinite minimization
`linprog`	Linear programming
`quadprog`	Quadratic programming

Equation Solving

Function	Purpose
`\`	Linear equation solving (see the online *MATLAB Function Reference* guide)
`fsolve`	Nonlinear equation solving
`fzero`	Scalar nonlinear equation solving

Least-Squares (Curve Fitting)

Function	Purpose
`\`	Linear least squares (see the online *MATLAB Function Reference* guide)
`lsqlin`	Constrained linear least squares
`lsqcurvefit`	Nonlinear curve fitting
`lsqnonlin`	Nonlinear least squares
`lsqnonneg`	Nonnegative linear least squares

Utility

Function	Purpose
`optimset`, `optimget`	Parameter setting

Demonstrations of Large-Scale Methods

Function	Purpose
`circustent`	Quadratic programming to find shape of a circus tent
`molecule`	Molecule conformation solution using unconstrained nonlinear minimization
`optdeblur`	Image deblurring using bounded linear least-squares

Demonstrations of Medium-Scale Methods

Function	Purpose
`bandemo`	Minimization of the `banana` function
`dfildemo`	Finite-precision filter design (requires *Signal Processing Toolbox*)
`goaldemo`	Goal attainment example
`optdemo`	Menu of demonstration routines
`tutdemo`	Tutorial walk-through

Function Arguments

These tables describe arguments used by Optimization Toolbox functions: the first describes input arguments, the second describes the output arguments, and the third describes the optimization options parameters structure `options`.

This table summarizes the input arguments. Not all functions use all arguments. See the individual reference pages for function-specific details about these arguments.

Table 4-1: Input Arguments

Argument	Description	Used by Functions
`A, b`	The matrix `A` and vector `b` are, respectively, the coefficients of the linear inequality constraints and the corresponding right-hand side vector: `A·x<=b`.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `linprog`, `lsqlin`, `quadprog`
`Aeq, beq`	The matrix `Aeq` and vector `beq` are, respectively, the coefficients of the linear equality constraints and the corresponding right-hand side vector: `Aeq·x=beq`.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `linprog`, `lsqlin`, `quadprog`
`C, d`	The matrix `C` and vector `d` are, respectively, the coefficients of the over- or under-determined linear system and the right-hand-side vector to be solved.	`lsqlin`, `lsqnonneg`
`f`	The vector of coefficients for the linear term in the linear equation `f'*x` or the quadratic equation `x'*H*x+f'*x`.	`linprog`, `quadprog`
`fun`	The function to be optimized. `fun` may be an inline object, or the name of an M-file, built-in function, or MEX-file. See the individual function reference pages for more information on `fun`.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `fzero`, `lsqcurvefit`, `lsqnonlin`

Table 4-1: Input Arguments (Continued)

Argument	Description	Used by Functions
`goal`	Vector of values that the objectives attempt to attain. The vector is the same length as the number of objectives.	`fgoalattain`
`H`	The matrix of coefficients for the quadratic terms in the quadratic equation `x'*H*x+f'*x`. `H` must be symmetric.	`quadprog`
`lb, ub`	Lower and upper bound vectors (or matrices). The arguments are normally the same size as `x`. However, if `lb` has fewer elements than `x`, say only `m`, then only the first `m` elements in `x` are bounded below; upper bounds in `ub` can be defined in the same manner. Unbounded variables may also be specified using `–Inf` (for lower bounds) or `Inf` (for upper bounds). For example, if `lb(i) = –Inf` then the variable `x(i)` is unbounded below.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`nonlcon`	The function that computes the nonlinear inequality and equality constraints. `nonlcon` is the name of an M-file or MEX-file. See the individual reference pages for more information on `nonlcon`.	`fgoalattain`, `fmincon`, `fminimax`
`ntheta`	The number of semi-infinite constraints.	`fseminf`
`options`	An optimization options parameter structure that defines parameters used by the optimization functions. For information about the parameters, see Table 4-3 or the individual function reference pages.	All functions

Table 4-1: Input Arguments (Continued)

Argument	Description	Used by Functions

`P1,P2,...`	Additional arguments to be passed to `fun`, `nonlcon` (if it exists), or `seminfcon` (if it exists), that is, when the optimization function calls the functions `fun`, `nonlcon`, or `seminfcon`, the calls are
	`f = feval(fun,x,P1,P2,...)`
	`[c, ceq] = feval(nonlcon,x,P1,P2,...)`
	`[c,ceq,K1,K2,...,Kn,s]= ...`
	`feval(seminfcon,x,s,P1,P2,...)`
	Using this feature, the same `fun` (or `nonlcon` or `seminfcon`) can solve a number of similar problems with different parameters, avoiding the need to use global variables.
		`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `fzero`, `lsqcurvefit`, `lsqnonlin`

`seminfcon`	The function that computes the nonlinear inequality and equality constraints *and* the semi-infinite constraints. `seminfcon` is the name of an M-file or MEX-file. See the individual function reference pages for `fseminf` more information on `seminfcon`.	`fseminf`
`weight`	A weighting vector to control the relative under-attainment or over-attainment of the objectives.	`fgoalattain`
`xdata`, `ydata`	The input data `xdata` and the observed output data `ydata` that is to be fit to an equation.	`lsqcurvefit`
`x0`	Starting point (a scalar, vector or matrix). (For `fzero`, `x0` can also be a two-element vector representing an interval that is known to contain a zero.)	All functions except `fminbnd`
`x1, x2`	The interval over which the function is minimized.	`fminbnd`

Table 4-2: Output Arguments

Argument	Description	Used by Functions
`attainfactor`	The attainment factor at the solution `x`.	`fgoalattain`
`exitflag`	The exit condition. For the meaning of a particular value, see the function reference pages.	All functions
`fval`	The value of the objective function `fun` at the solution `x`.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `fzero`, `linprog`, `quadprog`
`grad`	The value of the gradient of `fun` at the solution `x`.	`fmincon`, `fminunc`
`hessian`	The value of the Hessian of `fun` at the solution `x`.	`fmincon`, `fminunc`
`jacobian`	The value of the Jacobian of `fun` at the solution `x`.	`lsqcurvefit`, `lsqnonlin`, `fsolve`
`lambda`	The Lagrange multipliers at the solution `x`. `lambda` is a structure where each field is for a different constraint type. For structure field names, see individual function descriptions. (For `lsqnonneg`, `lambda` is simply a vector as `lsqnonneg` only handles one kind of constraint.)	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `lsqnonneg`, `quadprog`
`maxfval`	`max{fun(x)}` at the solution `x`.	`fminimax`
`output`	An output structure that contains information about the results of the optimization. For structure field names, see individual function descriptions.	All functions

Table 4-2: Output Arguments (Continued)

Argument	Description	Used by Functions
`residual`	The value of the residual at the solution `x`.	`lsqcurvefit`, `lsqlin`, `lsqnonlin`, `lsqnonneg`
`resnorm`	The value of the squared 2-norm of the residual at the solution `x`.	`lsqcurvefit`, `lsqlin`, `lsqnonlin`, `lsqnonneg`
`x`	The solution found by the optimization function. If `exitflag > 0` then `x` is a solution; otherwise, `x` is the value the optimization routine was at when it terminated prematurely.	All functions

This table describes fields in the optimization parameters structure, `options`. The column labeled L, M, B provides this information about each parameter:

- L – The parameter only applies to large-scale methods.
- M – The parameter only applies to medium-scale methods.
- B – The parameter applies to both large- and medium-scale methods.

Table 4-3: Optimization Options Parameters

Parameter Name	Description	L, M, B	Used by Functions
`DerivativeCheck`	Compare user-supplied analytic derivatives (gradients or Jacobian) to finite differencing derivatives.	M	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`Diagnostics`	Print diagnostic information about the function to be minimized or solved.	B	All but `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg`

Table 4-3: Optimization Options Parameters (Continued)

Parameter Name	Description	L, M, B	Used by Functions
`DiffMaxChange`	Maximum change in variables for finite difference derivatives.	M	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`DiffMinChange`	Minimum change in variables for finite difference derivatives.	M	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.	B	All
`GoalsExactAchieve`	Number of goals to achieve exactly (do not over- or underachieve).	M	`fgoalattain`
`GradConstr`	Gradients for the nonlinear constraints defined by user.	M	`fgoalattain`, `fmincon`, `fminimax`
`GradObj`	Gradient(s) for the objective function(s) defined by user.	B	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`
`Hessian`	Hessian for the objective function defined by user.	L	`fmincon`, `fminunc`
`HessPattern`	Sparsity pattern of the Hessian for finite differencing.	L	`fmincon`, `fminunc`
`HessUpdate`	Quasi-Newton updating scheme.	M	`fminunc`
`Jacobian`	Jacobian for the objective function defined by user.	B	`fsolve`, `lsqcurvefit`, `lsqnonlin`
`JacobPattern`	Sparsity pattern of the Jacobian for finite differencing.	L	`fsolve`, `lsqcurvefit`, `lsqnonlin`

Table 4-3: Optimization Options Parameters (Continued)

Parameter Name	Description	L, M, B	Used by Functions
`LargeScale`	Use large-scale algorithm if possible.	B	`fmincon`, `fminunc`, `fsolve`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`LevenbergMarquardt`	Chooses Levenberg-Marquardt over Gauss-Newton algorithm.	M	`fsolve`, `lsqcurvefit`, `lsqnonlin`
`LineSearchType`	Line search algorithm choice.	M	`fminunc`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`MaxFunEvals`	Maximum number of function evaluations allowed.	B	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`MaxIter`	Maximum number of iterations allowed.	B	All but `fzero` and `lsqnonneg`
`MaxPCGIter`	Maximum number of PCG iterations allowed.	L	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`MeritFunction`	Use goal attainment/minimax merit function (multiobjective) vs. `fmincon` (single objective).	M	`fgoalattain`, `fminimax`
`MinAbsMax`	Number of *F(x)* to minimize the worst case absolute values	M	`fminimax`
`PrecondBandWidth`	Upper bandwidth of preconditioner for PCG.	L	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`

Table 4-3: Optimization Options Parameters (Continued)

Parameter Name	Description	L, M, B	Used by Functions
`TolCon`	Termination tolerance on the constraint violation.	B	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`
`TolFun`	Termination tolerance on the function value.	B	All but `fminbnd`, `fzero`, and `lsqnonneg`
`TolPCG`	Termination tolerance on the PCG iteration.	L	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`TolX`	Termination tolerance on `x`.	B	All but `linprog` and `lsqlin`
`TypicalX`	Typical `x` values.	L	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`

fgoalattain

Purpose

Solve multiobjective goal attainment problem

$$\underset{x,\gamma}{\text{minimize}}\ \gamma \qquad \text{such that} \qquad \begin{aligned} &F(x) - weight \cdot \gamma \le goal \\ &c(x) \le 0 \\ &ceq(x) = 0 \\ &A \cdot x \le b \\ &Aeq \cdot x = beq \\ &lb \le x \le ub \end{aligned}$$

where x, *weight*, *goal*, b, *beq*, *lb*, and *ub* are vectors, A and *Aeq* are matrices, *c(x)*, *ceq(x)*, and *F(x)* are functions that return vectors. *F(x)*, *c(x)*, and *ceq(x)* can be nonlinear functions.

Syntax

```
x = fgoalattain(fun,x0,goal,weight)
x = fgoalattain(fun,x0,goal,weight,A,b)
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq)
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub)
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon)
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,...
                lb,ub,nonlcon,options)
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,...
                lb,ub,nonlcon,options,P1,P2,...)
[x,fval] = fgoalattain(...)
[x,fval,attainfactor] = fgoalattain(...)
[x,fval,attainfactor,exitflag] = fgoalattain(...)
[x,fval,attainfactor,exitflag,output] = fgoalattain(...)
[x,fval,attainfactor,exitflag,output,lambda] = fgoalattain(...)
```

Description

`fgoalattain` solves the goal attainment problem, which is one formulation for minimizing a multiobjective optimization problem.

`x = fgoalattain(fun,x0,goal,weight)` tries to make the objective functions supplied by `fun` attain the goals specified by `goal` by varying `x`, starting at `x0`, with weight specified by `weight`.

`x = fgoalattain(fun,x0,goal,weight,A,b)` solves the goal attainment problem subject to the linear inequalities `A*x <= b`.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq)` solves the goal attainment problem subject to the linear equalities `Aeq*x = beq` as well. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the goal attainment problem to the nonlinear inequalities `c(x)` or nonlinear equality constraints `ceq(x)` defined in `nonlcon`. `fgoalattain` optimizes such that `c(x) <= 0` and `ceq(x) = 0`. Set `lb=[]` and/or `ub=[]` if no bounds exist.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon,... options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon,... options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `nonlcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, `nonlcon`, and `options` if these arguments are not needed.

`[x,fval] = fgoalattain(...)` returns the values of the objective functions computed in `fun` at the solution `x`.

`[x,fval,attainfactor] = fgoalattain(...)` returns the attainment factor at the solution `x`.

`[x,fval,attainfactor,exitflag] = fgoalattain(...)` returns a value `exitflag` that describes the exit condition of `fgoalattain`.

`[x,fval,attainfactor,exitflag,output] = fgoalattain(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,attainfactor,exitflag,output,lambda] = fgoalattain(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `fgoalattain` are included below for `fun`, `goal`, `nonlcon`, `options`, `weight`, `attainfactor`, `exitflag`, `lambda`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a vector `F` of the objective functions evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
fun = inline('sin(x.*x)');
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function F = myfun(x)
F = ...      % Compute function values at x
```

To make an objective function as near as possible to a goal value, (i.e., neither greater than nor less than) set `options.GoalsExactAchieve` to the number of objectives required to be in the neighborhood of the goal values. Such objectives *must* be partitioned into the first elements of the vector `F` returned by `fun`.

If the gradient of the objective function can also be computed *and* `options.GradObj` is `'on'`, as set by

```
options = optimset('GradObj','on')
```

then the function `fun` must return, in the second output argument, the gradient value `G`, a matrix, at `x`. Note that by checking the value of `nargout` the function can avoid computing `G` when `'myfun'` is called with only one output argument (in the case where the optimization algorithm only needs the value of `F` but not `G`):

```
function [F,G] = myfun(x)
F = ...          % compute the function values at x
if nargout > 1   % two output arguments
   G = ...       % gradients evaluated at x
end
```

The gradient is the partial derivatives dF/dx of each `F` at the point `x`. If `F` is a vector of length `m` and `x` has length `n`, then the gradient `G` of `F(x)` is an n-by-m matrix where `G(i,j)` is the partial derivative of `F(j)` with respect to `x(i)` (i.e., the `j`th column of `G` is the gradient of the `j`th objective function `F(j)`).

`goal`	Vector of values that the objectives attempt to attain. The vector is the same length as the number of objectives `F` returned by `fun`. `fgoalattain` attempts to minimize the values in the vector `F` to attain the goal values given by `goal`.

`nonlcon` The function that computes the nonlinear inequality constraints `c(x) <=0` and nonlinear equality constraints `ceq(x)=0`. `nonlcon` is a string containing the name of a function (an M-file, a built-in, or a MEX-file). `nonlcon` takes a vector `x` and returns two arguments, a vector `c` of the nonlinear inequalities evaluated at `x` and a vector `ceq` of the nonlinear equalities evaluated at x. For example, if `nonlcon='mycon'` then the M-file `mycon.m` would have the form

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x
ceq = ...   % Compute the nonlinear equalities at x
```

If the gradients of the constraints can also be computed *and* `options.GradConstr` is `'on'`, as set by

```
options = optimset('GradConstr','on')
```

then the function `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`. Note that by checking the value of `nargout` the function can avoid computing `GC` and `GCeq` when `nonlcon` is called with only two output arguments (in the case where the optimization algorithm only needs the values of `c` and `ceq` but not `GC` and `GCeq`):

```
function [c,ceq,GC,GCeq] = mycon(x)
c = ...          % nonlinear inequalities at x
ceq = ...        % nonlinear equalities at x
if nargout > 2   % nonlcon called with 4 outputs
   GC = ...      % gradients of the inequalities
   GCeq = ...    % gradients of the equalities
end
```

If nonlcon returns a vector c of m components and x has length n, then the gradient GC of c(x) is an n-by-m matrix, where GC(i,j) is the partial derivative of c(j) with respect to x(i) (i.e., the jth column of GC is the gradient of the jth inequality constraint c(j)). Likewise, if ceq has p components, the gradient GCeq of ceq(x) is an n-by-p matrix, where GCeq(i,j) is the partial derivative of ceq(j) with respect to x(i) (i.e., the jth column of GCeq is the gradient of the jth equality constraint ceq(j)).

options Optimization parameter options. You can set or change the values of these parameters using the optimset function.

- DerivativeCheck – Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives.
- Diagnostics – Print diagnostic information about the function to be minimized or solved.
- DiffMaxChange – Maximum change in variables for finite-difference gradients.
- DiffMinChange – Minimum change in variables for finite-difference gradients.
- Display – Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' displays just the final output.
- GoalExactAchieve – Specifies the number of goals to "just" achieve, that is, do not try to over- or underachieve.
- GradConstr – Gradient for the constraints defined by user. See the description of nonlcon under the *Arguments* section above to see how to define the gradient in nonlcon.
- GradObj – Gradient for the objective function defined by user. See the description of fun under the *Arguments* section above to see how to define the gradient in fun. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method.
- MaxFunEvals – Maximum number of function evaluations allowed.

- `MaxIter` – Maximum number of iterations allowed.
- `MeritFunction` – Use goal attainment/minimax merit function if set to `'multiobj'`. Use `fmincon` merit function if set to `'singleobj'`.
- `TolCon` – Termination tolerance on the constraint violation.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

`weight` A weighting vector to control the relative under-attainment or over-attainment of the objectives in `fgoalattain`. When the values of goal are *all nonzero*, to ensure the same percentage of under- or over-attainment of the active objectives, set the weighting function to `abs(goal)`. (The active objectives are the set of objectives that are barriers to further improvement of the goals at the solution.)

Note: Setting `weight=abs(goal)` when any of the `goal` values are zero will cause that goal constraint to be treated like a hard constraint rather than as a goal constraint.

When the weighting function `weight` is positive, `fgoalattain` attempts to make the objectives less than the goal values. To make the objective functions greater than the goal values, set `weight` to be negative rather than positive. To make an objective function as near as possible to a goal value, use the `GoalsExactAchieve` parameter and put that objective as the first element of the vector returned by `fun` (see the description of `fun` and `options` above).

`attain-factor` `attainfactor` is the amount of over- or underachievement of the goals. If `attainfactor` is negative, the goals have been over-achieved; if `attainfactor` is positive, the goals have been under-achieved.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`
- `lambda.upper` for the upper bounds `ub`
- `lambda.ineqlin` for the linear inequalities
- `lambda.eqlin` for the linear equalities
- `lambda.ineqnonlin` for the nonlinear inequalities
- `lambda.eqnonlin` for the nonlinear equalities

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.

Examples

Consider a linear system of differential equations.

An output feedback controller, `K`, is designed producing a closed loop system

$$\begin{aligned}\dot{x} &= (A + BKC)x + Bu\\ y &= Cx\end{aligned}$$

The eigenvalues of the closed loop system are determined from the matrices `A`, `B`, `C`, and `K` using the command `eig(A+B*K*C)`. Closed loop eigenvalues must lie on the real axis in the complex plane to the left of the points `[–5,–3,–1]`. In order not to saturate the inputs, no element in `K` can be greater than `4` or be less than `–4`.

The system is a two-input, two-output, open loop, unstable system, with state-space matrices.

$$A = \begin{bmatrix} -0.5 & 0 & 0 \\ 0 & -2 & 10 \\ 0 & 1 & -2 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 \\ -2 & 2 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The set of *goal values* for the closed loop eigenvalues are initialized as

```
goal = [–5,–3,–1];
```

To ensure the same percentage of under- or over-attainment in the active objectives at the solution, the weighting matrix, `weight`, is set to `abs(goal)`.

Starting with a controller, `K = [–1,–1; –1,–1]`, first write an M-file, `eigfun.m`:

```
function F = eigfun(K,A,B,C)
F = sort(eig(A+B*K*C)); % Evaluate objectives
```

Next, enter system matrices and invoke an optimization routine:

```
A = [–0.5 0 0; 0 –2 10; 0 1 –2];
B = [1 0; –2 2; 0 1];
C = [1 0 0; 0 0 1];
K0 = [–1 –1; –1 –1];       % Initialize controller matrix
goal = [–5 –3 –1];         % Set goal values for the eigenvalues
weight = abs(goal)         % Set weight for same percentage
lb = –4*ones(size(K0));    % Set lower bounds on the controller
ub = 4*ones(size(K0));     % Set upper bounds on the controller
options = optimset('Display','iter');   % Set display parameter
[K,fval,attainfactor] = fgoalattain('eigfun',K0,...
    goal,weight,[],[],[],[],lb,ub,[],options,A,B,C)
```

This example can be run by using the demonstration script `goaldemo`. After about 12 iterations, a solution is

```
Active constraints:
     1
     2
     4
     9
    10
K =
    -4.0000    -0.2564
    -4.0000    -4.0000

fval =
    -6.9313
    -4.1588
    -1.4099

attainfactor =
    -0.3863
```

Discussion

The attainment factor indicates that each of the objectives has been over-achieved by at least 38.63% over the original design goals. The active constraints, in this case constraints 1 and 2, are the objectives that are barriers to further improvement and for which the percentage of over-attainment is met exactly. Three of the lower bound constraints are also active.

In the above design, the optimizer tries to make the objectives less than the goals. For a worst case problem where the objectives must be as near to the goals as possible, set `options.GoalsExactAchieve` to the number of objectives for which this is required.

Consider the above problem when you want all the eigenvalues to be equal to the goal values. A solution to this problem is found by invoking `fgoalattain` with `options.GoalsExactAchieve` set to `3`.

```
options = optimset('GoalsExactAchieve',3);
[K,fval,attainfactor] = fgoalattain(...
   eigfun,K0,goal,weight,[],[],[],[],lb,ub,[],options,A,B,C)
```

After about seven iterations, a solution is

```
K =
    –1.5954    1.2040
    –0.4201   –2.9046

fval =
    –5.0000
    –3.0000
    –1.0000

attainfactor =
     1.0859e–20
```

In this case the optimizer has tried to match the objectives to the goals. The attainment factor (of 1.0859e–20) indicates that the goals have been matched almost exactly.

Notes

This problem has discontinuities when the eigenvalues become complex; this explains why the convergence is slow. Although the underlying methods assume the functions are continuous, the method is able to make steps toward the solution since the discontinuities do not occur at the solution point. When the objectives and goals are complex, `fgoalattain` tries to achieve the goals in a least-squares sense.

Algorithm

Multiobjective optimization concerns the minimization of a set of objectives simultaneously. One formulation for this problem, and implemented in `fgoalattain`, is the goal attainment problem of Gembicki[1]. This entails the construction of a set of *goal* values for the objective functions. Multiobjective optimization is discussed fully in the *Introduction to Algorithms* chapter of this toolbox.

In this implementation, the slack variable γ is used as a dummy argument to minimize the vector of objectives *F(x)* simultaneously; *goal* is a set of values that the objectives attain. Generally, prior to the optimization, it is unknown whether the objectives will even reach the goals (under attainment) or be minimized less than the goals (over attainment). A weighting vector, *weight*, controls the relative under-attainment or over-attainment of the objectives.

`fgoalattain` uses a Sequential Quadratic Programming (SQP) method, which is described fully in the *Introduction to Algorithms* chapter. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [5] and [6]) is used together with the merit function proposed by [2, 3]. The line search is terminated when either merit function shows improvement. A modified Hessian, which takes advantage of special structure of this problem, is also used (see [5] and [6]). A full description of the modifications used is found in the "Goal Attainment Method" section of the *Introduction to Algorithms* chapter. Setting `options.MeritFunction = 'singleobj'` uses the merit function and Hessian used in `fmincon`.

`attainfactor` contains the value of γ at the solution. A negative value of γ indicates over-attainment in the goals.

See also the "SQP Implementation" section in the *Introduction to Algorithms* chapter for more details on the algorithm used and the types of procedures printed under the `Procedures` heading for the `options.Display = 'iter'` setting.

Limitations

The objectives must be continuous. `fgoalattain` may give only local solutions.

See Also

`fmincon`, `fminimax`, `optimset`

References

[1] Gembicki, F.W., "Vector Optimization for Control with Performance and Parameter Sensitivity Indices," Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, OH, 1974.

[2] Han, S.P., "A Globally Convergent Method For Nonlinear Programming," *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

[3] Powell, M.J.D., "A Fast Algorithm for Nonlineary Constrained Optimization Calculations," *Numerical Analysis*, ed. G.A. Watson, *Lecture Notes in Mathematics*, Springer Verlag, Vol. 630, 1978.

[4] Fleming, P.J. and A.P. Pashkevich, *Computer Aided Control System Design Using a Multi-Objective Optimisation Approach*, Control 1985 Conference, Cambridge, UK, pp. 174-179.

[5] Brayton, R.K., S.W. Director, G.D. Hachtel, and L.Vidigal, "A New Algorithm for Statistical Circuit Design Based on Quasi–Newton Methods and

Function Splitting," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, pp. 784–794, Sept. 1979.

[6] Grace, A.C.W., "Computer–Aided Control System Design Using Optimization Techniques," Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

fminbnd

Purpose

Find the minimum of a function of one variable on a fixed interval

$$\min_x f(x) \qquad \text{such that} \qquad x_1 < x < x_2$$

where x, x_1, and x_2 are scalars and *f(x)* is a function that returns a scalar.

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
x = fminbnd(fun,x1,x2,options,P1,P2,...)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the scalar valued function that is described in `fun` in the interval `x1 < x < x2`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fminbnd(fun,x1,x2,options,P1,P2,...)` provides for additional arguments, `P1`, `P2`, etc., which are passed to the objective function, `fun`. Use `options=[]` as a placeholder if no options are set.

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at the solution `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`.

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to `fminbnd` are included below for `fun`, `options`, `exitflag`, and `output`.

`fun` The function to be minimized. `fun` takes a scalar `x` and returns a scalar value `f` of the objective function evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
x = fminbnd(inline('sin(x*x)'),x0)
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolX` – Termination tolerance on `x`.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.algorithm` – The algorithm used.
- `output.funcCount` – The number of function evaluations.

Examples

A minimum of sin*(x)* occurs at

```
x = fminbnd('sin',0,2*pi)
x =
     4.7124
```

The value of the function at the minimum is

```
y = sin(x)
y =
    –1.0000
```

To find the minimum of the function

$$f(x) = (x-3)^2 - 1$$

on the interval (0,5), first write an M-file:

```
function f = myfun(x)
f = (x–3).^2 – 1;
```

Next, call an optimization routine:

```
x = fminbnd('myfun',0,5)
```

This generates the solution

```
x =
     3
```

The value at the minimum is

```
y = f(x)
y =
    –1
```

Algorithm

`fminbnd` is an M-file. The algorithm is based on golden section search and parabolic interpolation. A Fortran program implementing the same algorithm is given in [1].

Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval. In such a case, `fmincon` often gives faster and more accurate solutions.

`fminbnd` only handles real variables.

See Also

`fminsearch`, `fmincon`, `fminunc`, `optimset`, `inline`

References

[1] Forsythe, G.E., M.A. Malcolm, and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice Hall, 1976.

fmincon

Purpose

Find the minimum of a constrained nonlinear multivariable function

$$\min_{x} f(x) \qquad \text{subject to} \qquad \begin{aligned} & c(x) \le 0 \\ & ceq(x) = 0 \\ & A \cdot x \le b \\ & Aeq \cdot x \le beq \\ & lb \le x \le ub \end{aligned}$$

where *x, b, beq, lb,* and *ub* are vectors, *A* and *Aeq* are matrices, *c(x)* and *ceq(x)* are functions that return vectors, and *f(x)* is a function that returns a scalar. *f(x)*, *c(x)*, and *ceq(x)* can be nonlinear functions.

Syntax

```
x = fmincon(fun,x0,A,b)
x = fmincon(fun,x0,A,b,Aeq,beq)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2, ...)
[x,fval] = fmincon(...)
[x,fval,exitflag] = fmincon(...)
[x,fval,exitflag,output] = fmincon(...)
[x,fval,exitflag,output,lambda] = fmincon(...)
[x,fval,exitflag,output,lambda,grad] = fmincon(...)
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)
```

Description

`fmincon` finds the constrained minimum of a scalar function of several variables starting at an initial estimate. This is generally referred to as *constrained nonlinear optimization* or *nonlinear programming*.

`x = fmincon(fun,x0,A,b)` starts at `x0` and finds a minimum `x` to the function described in `fun` subject to the linear inequalities `A*x <= b`. `x0` can be a scalar, vector, or matrix.

`x = fmincon(fun,x0,A,b,Aeq,beq)` minimizes `fun` subject to the linear equalities `Aeq*x = beq` as well as `A*x <= b`. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`. Set `Aeq=[]` and `beq=[]` if no equalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimization to the nonlinear inequalities `c(x)` or equalities `ceq(x)` defined in `nonlcon`. `fmincon` optimizes such that `c(x) <= 0` and `ceq(x) = 0`. Set `lb=[]` and/or `ub=[]` if no bounds exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `nonlcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, `nonlcon`, and `options` if these arguments are not needed.

`[x,fval] = fmincon(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fmincon(...)` returns a value `exitflag` that describes the exit condition of `fmincon`.

`[x,fval,exitflag,output] = fmincon(...)` returns a structure `output` with information about the optimization.

`[x,fval,exitflag,output,lambda] = fmincon(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

`[x,fval,exitflag,output,lambda,grad] = fmincon(...)` returns the value of the gradient of `fun` at the solution `x`.

`[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)` returns the value of the Hessian of `fun` at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to fmincon are included below for fun, nonlcon, options, exitflag, lambda, and output.

fun The function to be minimized. fun takes a vector x and returns a scalar value f of the objective function evaluated at x. You can specify fun to be an inline object. For example,

```
fun = inline('sin(x''*x)');
```

Alternatively, fun can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If fun='myfun' then the M-file function myfun.m would have the form

```
function f = myfun(x)
f = ...            % Compute function value at x
```

If the gradient of fun can also be computed *and* options.GradObj is 'on', as set by

```
options = optimset('GradObj','on')
```

then the function fun must return, in the second output argument, the gradient value g, a vector, at x. Note that by checking the value of nargout the function can avoid computing g when fun is called with only one output argument (in the case where the optimization algorithm only needs the value of f but not g):

```
function [f,g] = myfun(x)
f = ...         % compute the function value at x
if nargout > 1  % fun called with two output arguments
   g = ...      % compute the gradient evaluated at x
end
```

The gradient is the partial derivatives of f at the point x. That is, the ith component of g is the partial derivative of f with respect to the ith component of x.

If the Hessian matrix can also be computed *and* options.Hessian is 'on', i.e., options = optimset('Hessian','on'), then the function fun must return the Hessian value H, a symmetric matrix, at x in a third output argument. Note that by checking the value of nargout we can avoid computing H when fun is called with only one or two output arguments (in the case where the optimization algorithm only needs the values of f and g but not H):

```
function [f,g,H] = myfun(x)
f = ...     % Compute the objective function value at x
if nargout > 1   % fun called with two output arguments
   g = ...    % gradient of the function evaluated at x
   if nargout > 2
   H = ...    % Hessian evaluated at x
end
```

The Hessian matrix is the second partial derivatives matrix of f at the point x. That is, the (ith,jth) component of H is the second partial derivative of f with respect to x_i and x_j, $\partial^2 f / \partial x_i \partial x_j$. The Hessian is by definition a symmetric matrix.

nonlcon The function that computes the nonlinear inequality constraints c(x)<=0 and nonlinear equality constraints ceq(x)=0. nonlcon is a string containing the name of a function (an M-file, a built-in, or a MEX-file). nonlcon takes a vector x and returns two arguments, a vector c of the nonlinear inequalities evaluated at x and a vector ceq of the nonlinear equalities evaluated at x. For example, if nonlcon='mycon' then the M-file mycon.m would have the form

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x
ceq = ...   % Compute the nonlinear equalities at x
```

If the gradients of the constraints can also be computed *and* options.GradConstr is 'on', as set by

```
options = optimset('GradConstr','on')
```

then the function nonlcon must also return, in the third and fourth output arguments, GC, the gradient of c(x), and GCeq, the gradient of ceq(x). Note that by checking the value of nargout the function can avoid computing GC and GCeq when nonlcon is called with only two output arguments (in the case where the optimization algorithm only needs the values of c and ceq but not GC and GCeq):

```
function [c,ceq,GC,GCeq] = mycon(x)
c = ...          % nonlinear inequalities at x
ceq = ...        % nonlinear equalities at x
if nargout > 2   % nonlcon called with 4 outputs
   GC = ...      % gradients of the inequalities
   GCeq = ...    % gradients of the equalities
end
```

If nonlcon returns a vector c of m components and x has length n, then the gradient GC of c(x) is an n-by-m matrix, where GC(i,j) is the partial derivative of c(j) with respect to x(i) (i.e., the jth column of GC is the gradient of the jth inequality constraint c(j)). Likewise, if ceq has p components, the gradient GCeq of ceq(x) is an n-by-p matrix, where GCeq(i,j) is the partial derivative of ceq(j) with respect to x(i) (i.e., the jth column of GCeq is the gradient of the jth equality constraint ceq(j)).

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `fmincon`, the *gradient must be provided (*see the description of `fun` above to see how) or else the medium-scale algorithm will be used.

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `GradObj` – Gradient for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the gradient in `fun`. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolCon` – Termination tolerance on the constraint violation.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `Hessian` – Hessian for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the Hessian in `fun`.
- `HessPattern` – Sparsity pattern of the Hessian for finite-differencing. If it is not convenient to compute the sparse Hessian matrix `H` in `fun`, the large-scale method in `fmincon` can approximate `H` via sparse finite-differences (of the gradient) provided the *sparsity structure* of `H` — i.e., locations of the nonzeros — is supplied as the value for `HessPattern`. In the worst case, if the structure is unknown, you can set `HessPattern` to be a dense matrix and a full finite-difference approximation will be computed at each iteration (this is the default). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

Parameters used by the medium-scale algorithm only:

- `DerivativeCheck` – Compare user-supplied derivatives (gradients of the objective and constraints) to finite-differencing derivatives.
- `DiffMaxChange` – Maximum change in variables for finite-difference gradients.
- `DiffMinChange` – Minimum change in variables for finite-difference gradients.
- `LineSearchType` – Line search algorithm choice.

`exitflag`	Describes the exit condition: • `> 0` indicates that the function converged to a solution `x`. • `0` indicates that the maximum number of function evaluations or iterations was reached. • `< 0` indicates that the function did not converge to a solution.
`lambda`	A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type): • `lambda.lower` for the lower bounds `lb`. • `lambda.upper` for the upper bounds `ub`. • `lambda.ineqlin` for the linear inequalities. • `lambda.eqlin` for the linear equalities. • `lambda.ineqnonlin` for the nonlinear inequalities. • `lambda.eqnonlin` for the nonlinear equalities.
`output`	A structure whose fields contain information about the optimization: • `output.iterations` – The number of iterations taken. • `output.funcCount` – The number of function evaluations. • `output.algorithm` – The algorithm used. • `output.cgiterations` – The number of PCG iterations (large-scale algorithm only). • `output.stepsize` – The final step size taken (medium-scale algorithm only). • `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Examples

Find values of x that minimize $f(x) = -x_1 x_2 x_3$, starting at the point `x = [10; 10; 10]` and subject to the constraints

$$0 \le x_1 + 2x_2 + 2x_3 \le 72$$

First, write an M–file that returns a scalar value `f` of the function evaluated at `x`:

```
function f = myfun(x)
f = –x(1) * x(2) * x(3);
```

Then rewrite the constraints as both less than or equal to a constant,

$$-x_1 - 2x_2 - 2x_3 \le 0$$

$$x_1 + 2x_2 + 2x_3 \le 72$$

Since both constraints are linear, formulate them as the matrix inequality $A \cdot x \le b$ where

$$A = \begin{bmatrix} -1 & -2 & -2 \\ 1 & 2 & 2 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 72 \end{bmatrix}$$

Next, supply a starting point and invoke an optimization routine:

```
x0 = [10; 10; 10];    % Starting guess at the solution
[x,fval] = fmincon('myfun',x0,A,b)
```

After 66 function evaluations, the solution is

```
x =
    24.0000
    12.0000
    12.0000
```

where the function value is

```
fval =
    –3.4560e+03
```

and linear inequality constraints evaluate to be `<= 0`

```
A*x–b=
    –72
      0
```

Notes

Large-scale optimization. To use the large-scale method, the gradient must be provided in `fun` (and `options.GradObj` set to `'on'`). A warning is given if no gradient is provided and `options.LargeScale` is not `'off'`. `fmincon` permits $g(x)$ to be an approximate gradient but this option is not recommended: the numerical behavior of most optimization codes is considerably more robust when the true gradient is used.

The large-scale method in `fmincon` is most effective when the matrix of second derivatives, i.e., the Hessian matrix $H(x)$, is also computed. However, evaluation of the true Hessian matrix is not required. For example, if you can supply the Hessian sparsity structure (using the `HessPattern` parameter in `options`), then `fmincon` will compute a sparse finite-difference approximation to $H(x)$.

If `x0` is not strictly feasible, `fmincon` chooses a new strictly feasible (centered) starting point.

If components of x have no upper (or lower) bounds, then `fmincon` prefers that the corresponding components of `ub` (or `lb`) be set to `Inf` (or `–Inf` for `lb`) as opposed to an arbitrary but very large positive (or negative in the case of lower bounds) number.

Several aspects of linearly constrained minimization should be noted:

- A dense (or fairly dense) column of matrix `Aeq` can result in considerable fill and computational cost.
- `fmincon` removes (numerically) linearly dependent rows in `Aeq`; however, this process involves repeated matrix factorizations and therefore can be costly if there are many dependencies.
- Each iteration involves a sparse least-squares solve with matrix

  $$B = Aeq^T R^{-T}$$

  where `R`^{T} is the Cholesky factor of the preconditioner. Therefore, there is a potential conflict between choosing an effective preconditioner and minimizing fill in B.

Medium-scale optimization. Better numerical results are likely if you specify equalities explicitly using `Aeq` and `beq`, instead of implicitly using `lb` and `ub`.

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, `'dependent'` is printed under the `Procedures` heading (when output is asked for using `options.Display = 'iter'`). The dependent equalities are only removed when the equalities are consistent. If the system of equalities is not consistent, the subproblem is infeasible and `'infeasible'` is printed under the `Procedures` heading.

Algorithm

Large-scale optimization. By default `fmincon` will choose the large-scale algorithm *if* the user supplies the gradient in `fun` (and `GradObj` is `'on'` in `options`) *and* if *only* upper and lower bounds exists *or only* linear equality constraints exist. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [5],[6]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. `fmincon` uses a Sequential Quadratic Programming (SQP) method. In this method, a Quadratic Programming (QP) subproblem is solved at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula (see `fminunc`, references [3, 6]).

A line search is performed using a merit function similar to that proposed by [1] and [2, 3]. The QP subproblem is solved using an active set strategy similar to that described in [4]. A full description of this algorithm is found in the "Constrained Optimization" section of the *Introduction to Algorithms* chapter of the toolbox manual.

See also the SQP implementation section in the *Introduction to Algorithms* chapter for more details on the algorithm used.

Diagnostics

Large-scale optimization. The large-scale code will not allow equal upper and lower bounds. For example if `lb(2)==ub(2)`, then `fmincon` gives the error:

```
Equal upper and lower bounds not permitted in this large-scale
method.
Use equality constraints and the medium-scale method instead.
```

If you only have equality constraints you can still use the large-scale method. But if you have both equalities and bounds, you must use the medium-scale method.

Limitations

The function to be minimized and the constraints must both be continuous. `fmincon` may only give local solutions.

When the problem is infeasible, `fmincon` attempts to minimize the maximum constraint value.

The objective function and constraint function must be real-valued, that is they cannot return complex values.

Large-scale optimization. To use the large-scale algorithm, the user must supply the gradient in `fun` (and `GradObj` must be set `'on'` in `options`), and only upper and lower bounds constraints may be specified, *or only* linear equality constraints must exist and `Aeq` cannot have more rows than columns. `Aeq` is typically sparse. See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

Currently, if the analytical gradient is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic gradient to the finite-difference gradient. Instead, use the medium-scale method to check the derivative with `options` parameter `MaxIter` set to 0 iterations. Then run the problem with the large-scale method.

References

[1] Han, S.P., “A Globally Convergent Method for Nonlinear Programming,” *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

[2] Powell, M.J.D., “The Convergence of Variable Metric Methods For Nonlinearly Constrained Optimization Calculations,” *Nonlinear Programming 3*, (O.L. Mangasarian, R.R. Meyer, and S.M. Robinson, eds.) Academic Press, 1978.

[3] Powell, M.J.D., “A Fast Algorithm for Nonlineary Constrained Optimization Calculations,” *Numerical Analysis*, ed. G.A. Watson, *Lecture Notes in Mathematics*, Springer Verlag, Vol. 630, 1978.

[4] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization,* Academic Press, London, 1981.

[5] Coleman, T.F. and Y. Li, “On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds,” *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.

[6] Coleman, T.F. and Y. Li, “An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds,” *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

See Also `fminbnd`, `fminsearch`, `fminunc`, `optimset`

Purpose

Solve the minimax problem

$$\min_{x} \max_{\{F_i\}} \{F_i(x)\} \quad \text{such that} \quad \begin{array}{l} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub \end{array}$$

where *x*, *b*, *beq*, *lb*, and *ub* are vectors, *A* and *Aeq* are matrices, *c(x)*, *ceq(x)*, and *F(x)* are functions that return vectors. *F(x)*, *c(x)*, and *ceq(x)* can be nonlinear functions.

Syntax

```
x = fminimax(fun,x0)
x = fminimax(fun,x0,A,b)
x = fminimax(fun,x0,A,b,Aeq,beq)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2,...)
[x,fval] = fminimax(...)
[x,fval,maxfval] = fminimax(...)
[x,fval,maxfval,exitflag] = fminimax(...)
[x,fval,maxfval,exitflag,output] = fminimax(...)
[x,fval,maxfval,exitflag,output,lambda] = fminimax(...)
```

Description

`fminimax` minimizes the worst-case value of a set of multivariable functions, starting at an initial estimate. The values may be subject to constraints. This is generally referred to as the *minimax* problem.

`x = fminimax(fun,x0)` starts at `x0` and finds a minimax solution `x` to the functions described in `fun`.

`x = fminimax(fun,x0,A,b)` solves the minimax problem subject to the linear inequalities `A*x <= b`.

`x = fminimax(fun,x,A,b,Aeq,beq)` solves the minimax problem subject to the linear equalities `Aeq*x = beq` as well. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fminimax(fun,x,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimax problem to the nonlinear inequalities `c(x)` or equality constraints `ceq(x)` defined in `nonlcon`. `fminimax` optimizes such that `c(x) <= 0` and `ceq(x) = 0`. Set `lb=[]` and/or `ub=[]` if no bounds exist.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `nonlcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, `nonlcon`, and `options` if these arguments are not needed.

`[x,fval] = fminimax(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,maxfval] = fminimax(...)` returns the maximum function value at the solution `x`.

`[x,fval,maxfval,exitflag] = fminimax(...)` returns a value `exitflag` that describes the exit condition of `fminimax`.

`[x,fval,maxfval,exitflag,output] = fminimax(...)` returns a structure `output` with information about the optimization.

`[x,fval,maxfval,exitflag,output,lambda] = fminimax(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to `fminimax` are included below for `fun`, `nonlcon`, `options`, `exitflag`, `lambda`, `maxfval`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a vector `F` of the objective functions evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
fun = inline('sin(x.*x)');
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function F = myfun(x)
F = ...            % Compute function values at x
```

To minimize the worst case absolute values of any of the elements of the vector *F(x)* (i.e., min{max abs{*F(x)*} }), partition those objectives into the first elements of `F` and set `options.MinAbsMax` to be the number of such objectives.

If the gradient of the objective function can also be computed *and* `options.GradObj` is `'on'`, as set by

```
options = optimset('GradObj','on')
```

then the function `fun` must return, in the second output argument, the gradient value `G`, a matrix, at `x`. Note that by checking the value of `nargout` the function can avoid computing `G` when `'myfun'` is called with only one output argument (in the case where the optimization algorithm only needs the value of `F` but not `G`):

```
function [F,G] = myfun(x)
F = ...          % compute the function values at x
if nargout > 1   % two output arguments
   G = ...       % gradients evaluated at x
end
```

The gradient is the partial derivatives dF/dx of each `F` at the point `x`. If `F` is a vector of length `m` and `x` has length `n`, then the gradient `G` of `F(x)` is an n-by-m matrix where `G(i,j)` is the partial derivative of `F(j)` with respect to `x(i)` (i.e., the `j`th column of `G` is the gradient of the `j`th objective function `F(j)`).

nonlcon The function that computes the nonlinear inequality constraints `c(x) <=0` and nonlinear equality constraints `ceq(x)=0`. `nonlcon` is a string containing the name of a function (an M-file, a built-in, or a MEX-file). `nonlcon` takes a vector `x` and returns two arguments, a vector `c` of the nonlinear inequalities evaluated at `x` and a vector `ceq` of the nonlinear equalities evaluated at `x`. For example, if `nonlcon='mycon'` then the M-file `mycon.m` would have the form

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x
ceq = ...   % Compute the nonlinear equalities at x
```

If the gradients of the constraints can also be computed *and* `options.GradConstr` is `'on'`, as set by

```
options = optimset('GradConstr','on')
```

then the function `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`. Note that by checking the value of `nargout` the function can avoid computing `GC` and `GCeq` when `nonlcon` is called with only two output arguments (in the case where the optimization algorithm only needs the values of `c` and `ceq` but not `GC` and `GCeq`):

```
function [c,ceq,GC,GCeq] = mycon(x)
c = ...          % nonlinear inequalities at x
ceq = ...        % nonlinear equalities at x
if nargout > 2   % nonlcon called with 4 outputs
   GC = ...      % gradients of the inequalities
   GCeq = ...    % gradients of the equalities
end
```

If `nonlcon` returns a vector `c` of `m` components and `x` has length `n`, then the gradient `GC` of `c(x)` is an n-by-m matrix, where `GC(i,j)` is the partial derivative of `c(j)` with respect to `x(i)` (i.e., the `j`th column of `GC` is the gradient of the `j`th inequality constraint `c(j)`). Likewise, if `ceq` has `p` components, the gradient `GCeq` of `ceq(x)` is an n-by-p matrix, where `GCeq(i,j)` is the partial derivative of `ceq(j)` with respect to `x(i)` (i.e., the `j`th column of `GCeq` is the gradient of the `j`th equality constraint `ceq(j)`).

`options`	Optimization parameter options. You can set or change the values of these parameters using the `optimset` function.

- `DerivativeCheck` – Compare user-supplied derivatives (gradients of the objective or constraints) to finite-differencing derivatives.
- `Diagnostics` – Print diagnostic information about the function to be minimized or solved.
- `DiffMaxChange` – Maximum change in variables for finite-difference gradients.
- `DiffMinChange` – Minimum change in variables for finite-difference gradients.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `GradConstr` – Gradient for the constraints defined by user. See the description of `nonlcon` under the *Arguments* section above to see how to define the gradient in `nonlcon`.
- `GradObj` – Gradient for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the gradient in `fun`. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `MeritFunction` – Use goal attainment/minimax merit function if set to `'multiobj'`. Use `fmincon` merit function if set to `'singleobj'`.
- `MinAbsMax` – Number of *F(x)* to minimize the worst case absolute values.
- `TolCon` – Termination tolerance on the constraint violation.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`
- `lambda.upper` for the upper bounds `ub`
- `lambda.ineqlin` for the linear inequalities
- `lambda.eqlin` for the linear equalities
- `lambda.ineqnonlin` for the nonlinear inequalities
- `lambda.eqnonlin` for the nonlinear equalities

`maxfval` Maximum of the function values evaluated at the solution `x`, that is, `maxfval = max{fun(x)}`.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.

Examples

Find values of x that minimize the maximum value of

$$\left[\ f_1(x)\ ,\ f_2(x)\ ,\ f_3(x)\ ,\ f_4(x)\ ,\ f_5(x)\ \right]$$

where

$$f_1(x) = 2x_1^2 + x_2^2 - 48x_1 - 40x_2 + 304$$
$$f_2(x) = -x_2^2 - 3x_2^2$$
$$f_3(x) = x_1 + 3x_2 - 18$$
$$f_4(x) = -x_1 - x_2$$
$$f_5(x) = x_1 + x_2 - 8.$$

First, write an M-file that computes the five functions at `x`:

```
function f = myfun(x)
f(1)= 2*x(1)^2+x(2)^2–48*x(1)–40*x(2)+304;     %Objectives
f(2)= –x(1)^2 – 3*x(2)^2;
f(3)= x(1) + 3*x(2) –18;
f(4)= –x(1)– x(2);
f(5)= x(1) + x(2) – 8;
```

Next, invoke an optimization routine:

```
x0 = [0.1; 0.1];       % Make a starting guess at solution
[x,fval] = fminimax('myfun',x0)
```

After seven iterations, the solution is

```
x =
     4.0000
     4.0000
fval =
     0.0000  –64.0000  –2.0000  –8.0000  –0.0000
```

Notes

The number of objectives for which the worst case absolute values of `F` are minimized is set in `options.MinAbsMax`. Such objectives should be partitioned into the first elements of `F`.

For example, consider the above problem, which requires finding values of `x` that minimize the maximum absolute value of

$$\left[\ f_1(x)\ ,\ f_2(x)\ ,\ f_3(x)\ ,\ f_4(x)\ ,\ f_5(x)\ \right]$$

This is solved by invoking `fminimax` with the commands

```
x0 = [0.1; 0.1];   % Make a starting guess at the solution
options = optimset('MinAbsMax',5);   % Minimize absolute values
[x,fval] = fminimax(fun,x0,[],[],[],[],[],[],[],options);
```

After seven iterations, the solution is

```
x =
     4.9256
     2.0796
fval =
    37.2356 –37.2356 –6.8357 –7.0052 –0.9948
```

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, `'dependent'` is printed under the `Procedures` heading (when the output option is `options.Display='iter'`). The dependent equalities are only removed when the equalities are consistent. If the system of equalities is not consistent, the subproblem is infeasible and `'infeasible'` is printed under the `Procedures` heading.

Algorithm

`fminimax` uses a Sequential Quadratic Programming (SQP) method [3]. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [4] and [5]) is used together with the merit function proposed by [1] and [2]. The line search is terminated when either merit function shows improvement. A modified Hessian that takes advantage of special structure of this problem is also used. Setting `options.MeritFunction = 'singleobj'` uses the merit function and Hessian used in `fmincon`.

See also the SQP implementation section in the *Introduction to Algorithms* chapter for more details on the algorithm used and the types of procedures printed under the `Procedures` heading for `options.Display='iter'` setting.

Limitations

The function to be minimized must be continuous. `fminimax` may only give local solutions.

See Also

`optimset`, `fgoalattain`, `lsqnonlin`

References

[1] Han, S.P., “A Globally Convergent Method For Nonlinear Programming,” *J. of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

[2] Powell, M.J.D., “A Fast Algorithm for Nonlineary Constrained Optimization Calculations,” *Numerical Analysis*, ed. G.A. Watson, *Lecture Notes in Mathematics*, Springer Verlag, Vol. 630, 1978.

[3] Brayton, R.K., S.W. Director, G.D. Hachtel, and L.Vidigal, “A New Algorithm for Statistical Circuit Design Based on Quasi–Newton Methods and Function Splitting,” *IEEE Trans. Circuits and Systems*, Vol. CAS-26, pp. 784-794, Sept. 1979.

[4] Grace, A.C.W., “Computer–Aided Control System Design Using Optimization Techniques,” Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

[5] Madsen, K. and H. Schjaer-Jacobsen, “Algorithms for Worst Case Tolerance Optimization,” *IEEE Transactions of Circuits and Systems*, Vol. CAS-26, Sept. 1979.

fminsearch

Purpose

Find the minimum of an unconstrained multivariable function

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

where x is a vector and $f(x)$ is a function that returns a scalar.

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
x = fminsearch(fun,x0,options,P1,P2,...)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fminsearch(fun,x0,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Pass an empty matrix for `options` to use the default values for `options`.

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to `fminsearch` are included below for `fun`, `options`, `exitflag`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a scalar value `f` of the objective function evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
x = fminsearch(inline('sin(x''*x)'),x0)
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function.`fminsearch` uses these `options` parameters:

- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`output`	A structure whose fields contain information about the optimization: • `output.iterations` – The number of iterations taken. • `output.algorithm` – The algorithm used. • `output.funcCount` – The number of function evaluations.

Examples

Minimize the one-dimensional function `f(x) = sin(x) + 3`.

To use an M-file, i.e., `fun = 'myfun'`, create a file `myfun.m`:

```
function f = myfun(x)
f = sin(x) + 3;
```

Then call `fminsearch` to find a minimum of `fun` near `2`:

```
x = fminsearch('myfun',2)
```

To minimize the function `f(x) = sin(x) + 3` using an inline object:

```
f = inline('sin(x)+3');
x = fminsearch(f,2);
```

Algorithms

`fminsearch` uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients as in `fminunc`.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

`fminsearch` is generally less efficient than `fminunc` for problems of order greater than two. However, when the problem is highly discontinuous, `fminsearch` may be more robust.

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. `fminsearch` may only give local solutions.

`fminsearch` only minimizes over the real numbers, that is, *x* must only consist of real numbers and *f(x)* must only return real numbers. When *x* has complex variables, they must be split into real and imaginary parts.

Note: `fminsearch` is not the preferred choice for solving problems that are sums-of-squares, that is, of the form: $\min f(x) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + L$. Instead use the `lsqnonlin` function, which has been optimized for problems of this form.

See Also

`fminbnd`, `fminunc`, `optimset`, `inline`

References

[1] Lagarias, J.C., J.A. Reeds, M.H. Wright, P.E. Wright, "Convergence Properties of the Nelder-Mead Simplex Algorithm in Low Dimensions," to appear in the *SIAM Journal of Optimization*.

fminunc

Purpose

Find the minimum of an unconstrained multivariable function

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

where x is a vector and $f(x)$ is a function that returns a scalar.

Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(fun,x0,options,P1,P2,...)
[x,fval] = fminunc(...)
[x,fval,exitflag] = fminunc(...)
[x,fval,exitflag,output] = fminunc(...)
[x,fval,exitflag,output,grad] = fminunc(...)
[x,fval,exitflag,output,grad,hessian] = fminunc(...)
```

Description

`fminunc` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminunc(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix.

`x = fminunc(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fminunc(fun,x0,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Pass an empty matrix for `options` to use the default values for `options`.

`[x,fval] = fminunc(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminunc(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fminunc(...)` returns a structure `output` that contains information about the optimization.

[x,fval,exitflag,output,grad] = fminunc(...) returns in grad the value of the gradient of fun at the solution x.

[x,fval,exitflag,output,grad,hessian] = fminunc(...) returns in hessian the value of the Hessian of the objective function fun at the solution x.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to fminunc are included below for fun, options, exitflag, and output.

fun The function to be minimized. fun takes a vector x and returns a scalar value f of the objective function evaluated at x. You can specify fun to be an inline object. For example,

```
x = fminunc(inline('sin(x''*x)'),x0)
```

Alternatively, fun can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If fun='myfun' then the M-file function myfun.m would have the form

```
function f = myfun(x)
f = ...             % Compute function value at x
```

If the gradient of fun can also be computed *and* options.GradObj is 'on', as set by

```
options = optimset('GradObj','on')
```

then the function fun must return, in the second output argument, the gradient value g, a vector, at x. Note that by checking the value of nargout the function can avoid computing g when fun is called with only one output argument (in the case where the optimization algorithm only needs the value of f but not g):

```
function [f,g] = myfun(x)
f = ...          % compute the function value at x
if nargout > 1   % fun called with 2 output arguments
   g = ...       % compute the gradient evaluated at x
end
```

The gradient is the partial derivatives $\partial f / \partial x$ of f at the point x. That is, the ith component of g is the partial derivative of f with respect to the ith component of x.

If the Hessian matrix can also be computed *and* options.Hessian is 'on', i.e., options = optimset('Hessian','on'), then the function fun must return the Hessian value H, a symmetric matrix, at x in a third output argument. Note that by checking the value of nargout we can avoid computing H when fun is called with only one or two output arguments (in the case where the optimization algorithm only needs the values of f and g but not H):

```
function [f,g,H] = myfun(x)
f = ...     % Compute the objective function value at x
if nargout > 1   % fun called with two output arguments
   g = ...    % gradient of the function evaluated at x
   if nargout > 2
   H = ...    % Hessian evaluated at x
end
```

The Hessian matrix is the second partial derivatives matrix of f at the point x. That is, the (ith,jth) component of H is the second partial derivative of f with respect to x_i and x_j, $\partial^2 f / \partial x_i \partial x_j$. The Hessian is by definition a symmetric matrix.

options Optimization parameter options. You can set or change the values of these parameters using the optimset function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the LargeScale option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For fminunc, the *gradient must be provided (*see the description of fun above to see how) or else the medium-scale algorithm will be used.

- LargeScale – Use large-scale algorithm if possible when set to 'on'. Use medium-scale algorithm when set to 'off'.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `GradObj` – Gradient for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the gradient in `fun`. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `Hessian` – Hessian for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the Hessian in `fun`.
- `HessPattern` – Sparsity pattern of the Hessian for finite-differencing. If it is not convenient to compute the sparse Hessian matrix `H` in `fun`, the large-scale method in `fminunc` can approximate `H` via sparse finite-differences (of the gradient) provided the *sparsity structure* of `H` — i.e., locations of the nonzeros — is supplied as the value for `HessPattern`. In the worst case, if the structure is unknown, you can set `HessPattern` to be a dense matrix and a full finite-difference approximation will be computed at each iteration (this is the default). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.

- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

Parameters used by the medium-scale algorithm only:

- `DerivativeCheck` – Compare user-supplied derivatives (gradient) to finite-differencing derivatives.
- `DiffMaxChange` – Maximum change in variables for finite-difference gradients.
- `DiffMinChange` – Minimum change in variables for finite-difference gradients.
- `LineSearchType` – Line search algorithm choice.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.stepsize` – The final step size taken (medium-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality: the norm of the gradient at the solution `x`.

Examples

Minimize the function $f(x) = 3*x_1^2 + 2*x1*x2 + x_2^2$.

To use an M-file, i.e., `fun = 'myfun'`, create a file `myfun.m`:

```
function f = myfun(x)
f = 3*x(1)^2 + 2*x(1)*x(2) + x(2)^2;    % cost function
```

Then call `fminunc` to find a minimum of `'myfun'` near `[1,1]`:

```
x0 = [1,1];
[x,fval] = fminunc('myfun',x0)
```

After a couple of iterations, the solution, `x`, and the value of the function at `x`, `fval`, are returned:

```
x =
  1.0e–008 *
   –0.7914    0.2260
fval =
  1.5722e–016
```

To minimize this function with the gradient provided, modify the M-file `myfun.m` so the gradient is the second output argument

```
function [f,g] = myfun(x)
f = 3*x(1)^2 + 2*x(1)*x(2) + x(2)^2;    % cost function
if nargout > 1
   g(1) = 6*x(1)+2*x(2);
   g(2) = 2*x(1)+2*x(2);
end
```

and indicate the gradient value is available by creating an optimization options structure with `options.GradObj` set to `'on'` using `optimset`:

```
options = optimset('GradObj','on');
x0 = [1,1];
[x,fval] = fminunc('myfun',x0,options)
```

After several iterations the solution `x` and `fval`, the value of the function at `x`, are returned:

```
x =
  1.0e–015 *
   –0.6661         0
fval2 =
  1.3312e–030
```

To minimize the function `f(x) = sin(x) + 3` using an inline object

```
f = inline('sin(x)+3');
x = fminunc(f,4)
```

which returns a solution

```
x =
    4.7124
```

Notes

`fminunc` is not the preferred choice for solving problems that are sums-of-squares, that is, of the form: $\min f(x) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + L$. Instead use the `lsqnonlin` function, which has been optimized for problems of this form.

To use the large-scale method, the gradient must be provided in `fun` (and `options.GradObj` set to `'on'`). A warning is given if no gradient is provided and `options.LargeScale` is not `'off'`.

Algorithms

Large-scale optimization. By default `fminunc` will choose the large-scale algorithm if the user supplies the gradient in `fun` (and `GradObj` is `'on'` in `options`). This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [8],[9]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. `fminunc` with `options.LargeScale` set to `'off'` uses the BFGS Quasi-Newton method with a mixed quadratic and cubic line search procedure. This quasi-Newton method uses the BFGS [1-4] formula for updating the approximation of the Hessian matrix. The DFP [5,6,7] formula, which approximates the inverse Hessian matrix, is selected by setting `options. HessUpdate` to `'dfp'` (and `options.LargeScale` to `'off'`). A steepest descent method is selected by setting `options.HessUpdate` to `'steepdesc'` (and `options.LargeScale` to `'off'`), although this is not recommended.

The default line search algorithm, i.e., when `options.LineSearchType` is set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options.LineSearchType` to `'cubicpoly'`. This second method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. A full description of the algorithms is given in the *Introduction to Algorithms* chapter.

Limitations

The function to be minimized must be continuous.`fminunc` may only give local solutions.

`fminunc` only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

Large-scale optimization. To use the large-scale algorithm, the user must supply the gradient in `fun` (and `GradObj` must be set `'on'` in `options`). See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

Currently, if the analytical gradient is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic gradient to the finite-difference gradient. Instead, use the medium-scale method to check the derivative with `options` parameter `MaxIter` set to 0 iterations. Then run the problem again with the large-scale method.

See Also

`fminsearch`, `optimset`, `inline`

References

[1] Broyden, C.G., “The Convergence of a Class of Double–Rank Minimization Algorithms,” *J. Inst. Math. Applic*., Vol. 6, pp. 76–90, 1970.

[2] Fletcher, R.,“A New Approach to Variable Metric Algorithms,” *Computer Journal*, Vol. 13, pp. 317–322, 1970.

[3] Goldfarb, D., “A Family of Variable Metric Updates Derived by Variational Means,” *Mathematics of Computing*, Vol. 24, pp. 23–26, 1970.

[4] Shanno, D.F., “Conditioning of Quasi–Newton Methods for Function Minimization,” *Mathematics of Computing*, Vol. 24, pp. 647–656, 1970.

[5] Davidon, W.C., “Variable Metric Method for Minimization,” *A.E.C. Research and Development Report*, ANL-5990, 1959.

[6] Fletcher, R. and M.J.D. Powell, “A Rapidly Convergent Descent Method for Minimization,” *Computer J.*, Vol. 6, pp. 163–168, 1963.

[7] Fletcher, R., “Practical Methods of Optimization,” Vol. 1, *Unconstrained Optimization*, John Wiley and Sons, 1980.

[8] Coleman, T.F. and Y. Li, “On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds,” *Mathematical Programming*, Vol. 67, Number 2, pp. 189–224, 1994.

[9] Coleman, T.F. and Y. Li, “An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds,” *SIAM Journal on Optimization*, Vol. 6, pp. 418–445, 1996.

Purpose

Find minimum of a semi-infinitely constrained multivariable nonlinear function

$$\min_x f(x) \quad \text{subject to} \quad \begin{aligned} & c(x) \le 0, \\ & ceq(x) = 0 \\ & A \cdot x \le b \\ & Aeq \cdot x = beq \\ & lb \le x \le ub \\ & K_1(x, w_1) \le 0 \\ & K_2(x, w_2) \le 0 \\ & \dots \\ & K_n(x, w_n) \le 0 \end{aligned}$$

where x, b, beq, lb, and ub are vectors, A and Aeq are matrices, $c(x)$, $ceq(x)$, and $K_i(x,w_i)$ are functions that return vectors, and $f(x)$ is a function that returns a scalar. $f(x)$, $c(x)$, and $ceq(x)$ can be nonlinear functions. The vectors (or matrices) $K_i(x, w_i) \le 0$ are continuous functions of both x and an additional set of variables $w_1, w_2, \dots, w_n$. The variables $w_1, w_2, \dots, w_n$ are vectors of, at most, length two.

Syntax

```
x = fseminf(fun,x0,ntheta,seminfcon)
x = fseminf(fun,x0,ntheta,seminfcon,A,b)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub,options)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,...
            lb,ub,options,P1,P2,...)
[x,fval] = fseminf(...)
[x,fval,exitflag] = fseminf(...)
[x,fval,exitflag,output] = fseminf(...)
[x,fval,exitflag,output,lambda] = fseminf(...)
```

Description

`fseminf` finds the minimum of a semi-infinitely constrained scalar function of several variables, starting at an initial estimate. The aim is to minimize $f(x)$ so the constraints hold for all possible values of $w_i \in \Re^1$ (or $w_i \in \Re^2$). Since it is impossible to calculate all possible values of $K_i(x, w_i)$, a region must be chosen for w_i over which to calculate an appropriately sampled set of values.

`x = fseminf(fun,x0,ntheta,seminfcon)` starts at `x0` and finds a minimum of the function `fun` constrained by `ntheta` semi-infinite constraints defined in `seminfcon`.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b)` also tries to satisfy the linear inequalities `A*x <= b`.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq)` minimizes subject to the linear equalities `Aeq*x = beq` as well. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub,options, P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `seminfcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, and `options` if these arguments are not needed.

`[x,fval] = fseminf(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fseminf(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fseminf(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,exitflag,output,lambda] = fseminf(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `fseminf` are included below for `fun`, `ntheta`, `options`, `seminfcon`, `exitflag`, `lambda`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a scalar value `f` of the objective function evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
fun = inline('sin(x''*x)');
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function f = myfun(x)
f = ...            % Compute function value at x
```

If the gradient of `fun` can also be computed *and* `options.GradObj` is `'on'`, as set by

```
options = optimset('GradObj','on')
```

then the function `fun` must return, in the second output argument, the gradient value `g`, a vector, at `x`. Note that by checking the value of `nargout` the function can avoid computing `g` when `fun` is called with only one output argument (in the case where the optimization algorithm only needs the value of `f` but not `g`):

```
function [f,g] = myfun(x)
f = ...          % compute the function value at x
if nargout > 1   % fun called with 2 output arguments
   g = ...       % compute the gradient evaluated at x
end
```

The gradient is the partial derivatives of `f` at the point `x`. That is, the `i`th component of `g` is the partial derivative of `f` with respect to the `i`th component of `x`.

`ntheta` The number of semi-infinite constraints.

`options`	Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. • `DerivativeCheck` – Compare user-supplied derivatives (gradients) to finite-differencing derivatives. • `Diagnostics` – Print diagnostic information about the function to be minimized or solved. • `DiffMaxChange` – Maximum change in variables for finite-difference gradients. • `DiffMinChange` – Minimum change in variables for finite-difference gradients. • `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output. • `GradObj` – Gradient for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the gradient in `fun`. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method. • `MaxFunEvals` – Maximum number of function evaluations allowed. • `MaxIter` – Maximum number of iterations allowed. • `TolCon` – Termination tolerance on the constraint violation. • `TolFun` – Termination tolerance on the function value. • `TolX` – Termination tolerance on `x`.
`seminfcon`	The function that computes the vector of nonlinear inequality constraints, `c`, a vector of nonlinear equality constraints, `ceq`, and `ntheta` semi-infinite constraints (vectors or matrices) `K1`, `K2`,..., `Kntheta` evaluated over an interval `S` at the point `x`. `seminfcon` is a string containing the name of the function (an M-file, a built-in, or a MEX-file). For example, if `seminfcon='myinfcon'` then the M-file `myinfcon.m` would have the form

```
function [c,ceq,K1,K2,...,Kntheta,S] = myinfcon(x,S)
% Initial sampling interval
if isnan(S(1,1)),
   S = ...% S has ntheta rows and 2 columns
end
w1 = ...% Compute sample set
w2 = ...% Compute sample set
...
wntheta = ... % Compute sample set
K1 = ... % 1st semi-infinite constraint at x and w
K2 = ... % 2nd semi-infinite constraint at x and w
...
Kntheta = ...% last semi-infinite constraint at x and w
c = ...      % Compute nonlinear inequalities at x
ceq = ...    % Compute the nonlinear equalities at x
```

S is a recommended sampling interval, which may or may not be used. Return [] for c and ceq if no such constraints exist.

The vectors or matrices, K1, K2, ..., Kntheta, contain the semi-infinite constraints evaluated for a sampled set of values for the independent variables, w1, w2, ... wntheta, respectively. The two column matrix, S, contains a recommended sampling interval for values of w1, w2, ... wntheta, which are used to evaluate K1, K2, ... Kntheta. The ith row of S contains the recommended sampling interval for evaluating K*i*. When K*i* is a vector, use only S(i,1) (the second column can be all zeros). When K*i* is a matrix, s(i,2) is used for the sampling of the rows in K*i*, S(i,1) is used for the sampling interval of the columns of K*i* (see "Two-Dimensional Example" in the *Examples* section). On the first iteration S is NaN, so that some initial sampling interval must be determined by seminfcon.

exitflag Describes the exit condition:

- > 0 indicates that the function converged to a solution x.
- 0 indicates that the maximum number of function evaluations or iterations was reached.
- < 0 indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`.
- `lambda.upper` for the upper bounds `ub`.
- `lambda.ineqlin` for the linear inequalities.
- `lambda.eqlin` for the linear equalities.
- `lambda.ineqnonlin` for the nonlinear inequalities.
- `lambda.eqnonlin` for the nonlinear equalities.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.
- `output.stepsize` – The final step size taken.

Notes

The recommended sampling interval, `S`, set in `seminfcon`, may be varied by the optimization routine `fseminf` during the computation because values other than the recommended interval may be more appropriate for efficiency or robustness. Also, the finite region w_i, over which $K_i(x, w_i)$ is calculated, is allowed to vary during the optimization provided that it does not result in significant changes in the number of local minima in $K_i(x, w_i)$.

Examples

One-Dimensional Example

Find values of x that minimize

$$f(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 + (x_3 - 0.5)^2$$

where

$$K_1(x, w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 \le 1$$

$$K_2(x, w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1$$

for all values of w_1 and w_2 over the ranges

$$1 \le w_1 \le 100$$

$$1 \le w_2 \le 100$$

Note that the semi-infinite constraints are one-dimensional, that is, vectors. Since the constraints must be in the form $K_i(x, w_i) \le 0$ you will need to compute the constraints as

$$K_1(x, w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 - 1 \le 0$$

$$K_2(x, w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 - 1 \le 0$$

First, write an M-file that computes the objective function:

```
function f = myfun(x,s)
% Objective function
f = sum((x–0.2).^2);
```

Second, write an M-file, `mycon.m`, that computes the nonlinear equality and inequality constraints and the semi-infinite constraints:

```
function [c,ceq,K1,K2,s] = mycon(x,s)
% Initial sampling interval
if isnan(s(1,1)),
   s = [0.2 0; 0.2 0];
end
% Sample set
w1 = 1:s(1,1):100;
w2 = 1:s(2,1):100;

% Semi–infinite constraints
K1 = sin(w1*X(1)).*cos(w1*X(2)) – 1/1000*(w1–50).^2 –...
       sin(w1*X(3))–X(3)–1;
K2 = sin(w2*X(2)).*cos(w2*X(1)) – 1/1000*(w2–50).^2 –...
       sin(w2*X(3))–X(3)–1;

% No constraints
c = []; ceq=[];
```

```
% Plot a graph of semi-infinite constraints
plot(w1,K1,'-',w2,K2,':'),title('Semi–infinite constraints')
drawnow
```

Then, invoke an optimization routine:

```
x0 = [0.5; 0.2; 0.3];      % Starting guess at the solution
[x,fval] = fseminf('myfun',x0,2,'mycon')
```

After eight iterations, the solution is

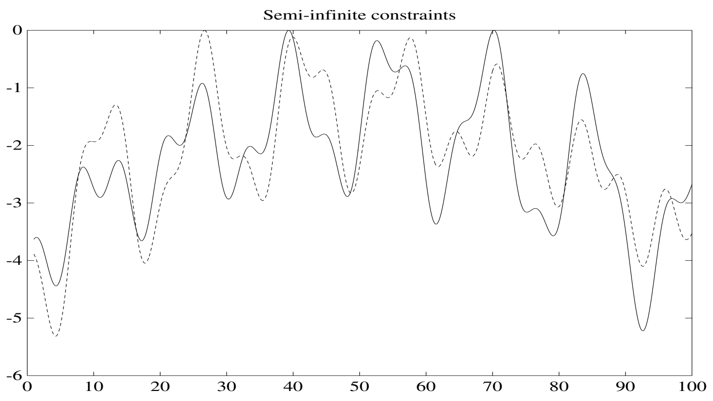
```
x =
     0.6673
     0.3013
     0.4023
```

The function value and the maximum values of the semi-infinite constraints at the solution x are

```
fval =
     0.0770

[c,ceq,K1,K2] = mycon(x,NaN);
max(K1)
ans =
     –0.0017
max(K2)
ans =
     –0.0845
```

A plot of the semi-infinite constraints is produced.



This plot shows how peaks in both functions are on the constraint boundary.

The plot command inside of `'mycon.m'` will slow down the computation; remove this line to improve the speed.

Two-Dimensional Example

Find values of x that minimize $f(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 + (x_3 - 0.5)^2$

where

$$K_1(x, w) = \sin(w_1 x_1)\cos(10 w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(10 w_1 x_3) - x_3 + \ldots$$
$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) + -x_3 \le 1.5$$

for all values of w_1 and w_2 over the ranges:

$1 \le w_1 \le 100$

$1 \le w_2 \le 100$

starting at the point $x = [0.2, 0.2, 0.2]$.

Note that the semi-infinite constraint is two-dimensional, that is, a matrix.

First, we reuse the file myfun.m for the objective function from the previous example.

Second, write an M-file for the constraints, called mycon.m:

```
function [c,ceq,K1,s] = mycon(x,s)
% Initial sampling interval
if isnan(s(1,1)),
   s = [2 2];
end
%
% Sampling set
w1x = 1:s(1,1):100;
w1y = 1:s(1,2):100;
[wx,wy] = meshgrid(w1x,w1y);
%
% Semi–infinite constraint
K1 = sin(wx*X(1)).*cos(wy*X(2))–1/1000*(wx–50).^2 –...
       sin(wx*X(3))–X(3)+sin(wy*X(2)).*cos(wx*X(1))–...
       1/1000*(wy–50).^2–sin(wy*X(3))–X(3)–1.5;
%
% No finite nonlinear constraints
c = []; ceq=[];
%
% Mesh plot
mesh(K1,title('Semi–infinite constraint')
drawnow
```

Next, invoke an optimization routine:

```
x0 = [0.25, 0.25, 0.25];    % Starting guess at the solution
[x,fval] = fseminf('myfun',x0,1,'mycon')
```

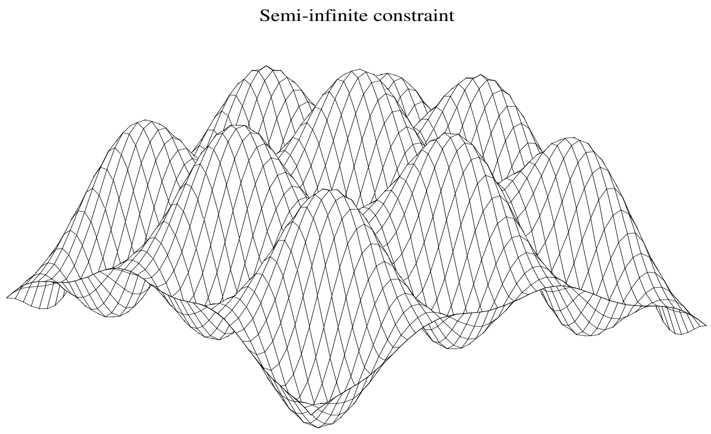
After seven iterations, the solution is

```
x =
     0.2081    0.2066    0.1965
[c,ceq,K1] = mycon(x,NaN);
```

The function value and the maximum value of the semi-infinite constraint at the solution x are

```
fval =
     1.2066e–04
max(max(K1))
ans =
    –0.0262
```

The following mesh plot is produced.



Algorithm

`fseminf` uses cubic and quadratic interpolation techniques to estimate peak values in the semi-infinite constraints. The peak values are used to form a set of constraints that are supplied to an SQP method as in the function `fmincon`. When the number of constraints changes, Lagrange multipliers are reallocated to the new set of constraints.

The recommended sampling interval calculation uses the difference between the interpolated peak values and peak values appearing in the data set to estimate whether more or fewer points need to be taken. The effectiveness of the interpolation is also taken into consideration by extrapolating the curve and comparing it to other points in the curve. The recommended sampling interval is decreased when the peak values are close to constraint boundaries, i.e., zero.

See also the SQP implementation section in the *Introduction to Algorithms* chapter for more details on the algorithm used and the types of procedures printed under the `Procedures` heading for `options.Display = 'iter'` setting.

Limitations

The function to be minimized, the constraints, and semi-infinite constraints, must be continuous functions of `x` and `w`. `fseminf` may only give local solutions.

When the problem is not feasible, `fseminf` attempts to minimize the maximum constraint value.

See Also

`fmincon`, `optimset`

Purpose

Solve a system of nonlinear equations

$$F(x) = 0$$

for x, where x is a vector and *F(x)* is a function that returns a vector value.

Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
x = fsolve(fun,x0,options,P1,P2, ... )
[x,fval] = fsolve(...)
[x,fval,exitflag] = fsolve(...)
[x,fval,exitflag,output] = fsolve(...)
[x,fval,exitflag,output,jacobian] = fsolve(...)
```

Description

`fsolve` finds a root (zero) of a system of nonlinear equations.

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations described in `fun`.

`x = fsolve(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fsolve(fun,x0,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Pass an empty matrix for `options` to use the default values for `options`.

`[x,fval] = fsolve(fun,x0)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fsolve(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fsolve(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,exitflag,output,jacobian] = fsolve(...)` returns the Jacobian of `fun` at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `fsolve` are included below for `fun`, `options`, `exitflag`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a vector `F` of the nonlinear equations evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
x = fsolve(inline('sin(x.*x)'),x0)
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function F = myfun(x)
F = ...             % Compute function values at x
```

If the Jacobian can also be computed *and* `options.Jacobian` is `'on'`, set by

```
options = optimset('Jacobian','on')
```

then the function `fun` must return, in a second output argument, the Jacobian value `J`, a matrix, at `x`. Note that by checking the value of `nargout` the function can avoid computing `J` when `fun` is called with only one output argument (in the case where the optimization algorithm only needs the value of `F` but not `J`):

```
function [F,J] = myfun(x)
F = ...           % objective function values at x
if nargout > 1    % two output arguments
   J = ...   % Jacobian of the function evaluated at x
end
```

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, then the Jacobian `J` is an m-by-n matrix where `J(i,j)` is the partial derivative of `F(i)` with respect to `x(j)`. (Note that the Jacobian `J` is the transpose of the gradient of `F`.)

`options`	Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `lsqnonlin`, the nonlinear system of equations cannot be underdetermined; that is, the number of equations (the number of elements of `F` returned by `fun`) must be at least as many as the length of `x` or else the medium-scale algorithm will be used.

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `Jacobian` – Jacobian for the objective function defined by user. See the description of `fun` above to see how to define the Jacobian in `fun`.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `JacobPattern` – Sparsity pattern of the Jacobian for finite-differencing. If it is not convenient to compute the Jacobian matrix `J` in `fun`, `lsqnonlin` can approximate `J` via sparse finite-differences provided the structure of `J` — i.e., locations of the nonzeros — is supplied as the value for `JacobPattern`. In the worst case, if the structure is unknown, you can set `JacobPattern` to be a dense matrix and a full finite-difference approximation will be computed in each iteration (this is the default if `JacobPattern` is not set). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

Parameters used by the medium-scale algorithm only:

- `DerivativeCheck` – Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
- `DiffMaxChange` – Maximum change in variables for finite-differencing.
- `DiffMinChange` – Minimum change in variables for finite-differencing.
- `LevenbergMarquardt` – Choose Levenberg-Marquardt over Gauss-Newton algorithm.
- `LineSearchType` – Line search algorithm choice.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.stepsize` – The final step size taken (medium-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Examples

Example 1: Find a zero of the system of two equations and two unknowns

$$2x_1 - x_2 = e^{-x_1}$$

$$-x_1 + 2x_2 = e^{-x_2}$$

Thus we want to solve the following system for x

$$2x_1 - x_2 - e^{-x_1} = 0$$

$$-x_1 + 2x_2 - e^{-x_2} = 0$$

starting at `x0 = [-5 -5]`.

First, write an M-file that computes `F`, the values of the equations at `x`:

```
function F = myfun(x)
F = [2*x(1) - x(2) - exp(-x(1));
      -x(1) + 2*x(2) - exp(-x(2))];
```

Next, call an optimization routine:

```
x0 = [−5; −5];       % Make a starting guess at the solution
options=optimset('Display','iter');   % Option to display output
[x,fval] = fsolve('myfun',x0,options)    % call optimizer
```

After 28 function evaluations, a zero is found:

```
                                          Norm of  First-order
Iteration  Func-count        f(x)           step   optimality  CG-iterations
        1           4      47071.2              1    2.29e+004             0
        2           7      6527.47        1.45207    3.09e+003             1
        3          10      918.372        1.49186          418             1
        4          13       127.74        1.55326         57.3             1
        5          16      14.9153        1.57591         8.26             1
        6          19     0.779051        1.27662         1.14             1
        7          22   0.00372453       0.484658       0.0683             1
        8          25  9.21617e-008     0.0385552     0.000336             1
        9          28  5.66133e-017   0.000193707    8.34e-009             1

  Optimization terminated successfully:
   Relative function value changing by less than OPTIONS.TolFun
  x =
      0.5671
      0.5671
  fval =
     1.0e−08 *
      −0.5320
      −0.5320
```

Example 2: Find a matrix x that satisfies the equation

$$X * X * X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix},$$

starting at the point `x= [1,1; 1,1]`.

First, write an M-file that computes the equations to be solved:

```
function F = myfun(x)
F = x*x*x–[1,2;3,4];
```

Next, invoke an optimization routine:

```
x0 = ones(2,2);  % Make a starting guess at the solution
options = optimset('Display','off'); % Turn off Display
[x,Fval,exitflag] = fsolve('myfun',x0,options)
```

The solution is

```
x =
    –0.1291    0.8602
     1.2903    1.1612
Fval =
     1.0e–03 *
     0.1541   –0.1163
     0.0109   –0.0243
exitflag =
     1
```

and the residual is close to zero

```
sum(sum(Fval.*Fval))
ans =
     3.7974e–008
```

Notes

If the system of equations is linear, then \ (the backslash operator: see `help slash`) should be used for better speed and accuracy. For example, say we want to find the solution to the following linear system of equations:

$$\begin{aligned} 3x_1 + 11x_2 - 2x_3 &= 7 \\ x_1 + x_2 - 2x_3 &= 4 \\ x_1 - x_2 + x_3 &= 19 \end{aligned}$$

Then the problem is formulated and solved as

```
A = [ 3 11 -2; 1 1 -2; 1 -1 1];
b = [ 7; 4; 19];
x = A\b
x =
   13.2188
   -2.3438
    3.4375
```

Algorithm

The methods are based on the nonlinear least-squares algorithms also used in `lsqnonlin`. The advantage of using a least-squares method is that if the system of equations is never zero due to small inaccuracies, or because it just does not have a zero, the algorithm still returns a point where the residual is small. However, if the Jacobian of the system is singular, the algorithm may converge to a point that is not a solution of the system of equations (see *Limitations* and *Diagnostics* below).

Large-scale optimization. By default `fsolve` will choose the large-scale algorithm. The algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [5],[6]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. `fsolve` with `options.LargeScale` set to `'off'` uses the Gauss-Newton method [4] with line-search. Alternatively, a Levenberg-Marquardt method [1], [2], [3] with line-search may be selected. The choice of algorithm is made by setting `options.LevenbergMarquardt`. Setting `options.LevenbergMarquardt` to `'on'` (and `options.LargeScale` to `'off'`) selects the Levenberg-Marquardt method.

The default line search algorithm, i.e., `options.LineSearchType` set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options.LineSearchType` to `'cubicpoly'`. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. The algorithms used are described fully in the *Introduction to Algorithms* chapter.

Diagnostics

`fsolve` may converge to a nonzero point and give this message

```
Optimizer is stuck at a minimum that is not a root
Try again with a new starting guess
```

In this case, run `fsolve` again with other starting values.

Limitations

The function to be solved must be continuous. When successful, `fsolve` only gives one root. `fsolve` may converge to a nonzero point, in which case, try other starting values.

`fsolve` only handles real variables. When x has complex variables, the variables must be split into real and imaginary parts.

Large-scale optimization. Currently, if the analytical Jacobian is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic Jacobian to the finite-difference Jacobian. Instead, use the medium-scale method to check the derivative with `options` parameter `MaxIter` set to 0 iterations. Then run the problem again with the large-scale method. See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

The preconditioner computation used in the preconditioned conjugate gradient part of the large-scale method forms J^TJ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product J^TJ, may lead to a costly solution process for large problems.

See Also

`\`, `optimset`, `lsqnonlin`, `lsqcurvefit`, `inline`

References

[1] Levenberg, K., “A Method for the Solution of Certain Problems in Least Squares,” *Quarterly Applied Mathematics 2*, pp. 164-168, 1944.

[2] Marquardt, D., “An Algorithm for Least–squares Estimation of Nonlinear Parameters,” *SIAM Journal Applied Mathematics*, Vol. 11, pp. 431-441, 1963.

[3] More, J. J., “The Levenberg–Marquardt Algorithm: Implementation and Theory,” *Numerical Analysis*, ed. G. A. Watson, *Lecture Notes in Mathematics* 630, Springer Verlag, pp. 105-116, 1977.

[4] Dennis, J. E. Jr., “Nonlinear Least Squares,” *State of the Art in Numerical Analysis*, ed. D. Jacobs, Academic Press, pp. 269-312.

[5] Coleman, T.F. and Y. Li, “On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds,” *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.

[6] Coleman, T.F. and Y. Li, “An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds,” *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

Purpose

Zero of a function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
x = fzero(fun,x0,options,P1,P2,...)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

Description

`x = fzero(fun,x0)` tries to find a zero of `fun` near `x0` if `x0` is a scalar. The value `x` returned by `fzero` is near a point where `fun` changes sign, or `NaN` if the search fails. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.

If `x0` is a vector of length two, `fzero` assumes `x0` is an interval where the sign of `fun(x0(1))` differs from the sign of `fun(x0(2))`. An error occurs if this is not true. Calling `fzero` with such an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`x = fzero(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fzero(fun,x0,options,P1,P2,...)` provides for additional arguments, `P1`, `P2`, etc., which are passed to the objective function, `fun`. Use `options=[]` as a placeholder if no options are set.

`[x,fval] = fzero(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fzero(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fzero(...)` returns a structure `output` that contains information about the optimization.

Note: For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the *x*-axis.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `fzero` are included below for `fun`, `options`, `exitflag`, and `output`.

`fun` The function to be minimized. `fun` takes a scalar `x` and returns a scalar value `f` of the objective function evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
x = fzero(inline('sin(x*x)'),x0)
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. `fzero` uses these `options` structure fields:

- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `TolX` – Termination tolerance on `x`.

`exitflag` Describes the exit condition:

- `> 0` indicates that `fzero` found a zero `x`
- `< 0` then no interval was found with a sign change, or a `NaN` or `Inf` function value was encountered during the search for an interval containing a sign change, or a complex function value was encountered during the search for an interval containing a sign change.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken (for `fzero`, this is the same as the number of function evaluations).
- `output.algorithm` – The algorithm used.
- `output.funcCount` – The number of function evaluations.

Examples Calculate π by finding the zero of the sine function near 3.

```
x = fzero('sin',3)
x =
    3.1416
```

To find the zero of cosine between 1 and 2:

```
x = fzero('cos',[1 2])
x =
    1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

To find a zero of the function

$$f(x) = x^3 - 2x - 5$$

write an M-file called `f.m`.

```
function y = f(x)
y = x.^3–2*x–5;
```

To find the zero near 2

```
z = fzero('f',2)
z =
    2.0946
```

Since this function is a polynomial, the statement `roots([1 0 –2 –5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
  –1.0473 + 1.1359i
  –1.0473 – 1.1359i
```

Notes

Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

Algorithm

The `fzero` command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

Limitations

The `fzero` command defines a *zero* as a point where the function crosses the x-axis. Points where the function touches, but does not cross, the x-axis are not valid zeros. For example, `y = x.^2` is a parabola that touches the x-axis at 0. Since the function never crosses the x-axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

See Also

`roots`, `fminbnd`, `fsolve`, `\`, `inline`, `optimset`

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

Purpose

Solve a linear programming problem

$$\min_x f^T x \qquad \text{such that} \qquad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

where f, x, b, beq, lb, and ub are vectors and A and Aeq are matrices.

Syntax

```
x = linprog(f,A,b,Aeq,beq)
x = linprog(f,A,b,Aeq,beq,lb,ub)
x = linprog(f,A,b,Aeq,beq,lb,ub,x0)
x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)
[x,fval] = linprog(...)
[x,fval,exitflag] = linprog(...)
[x,fval,exitflag,output] = linprog(...)
[x,fval,exitflag,output,lambda] = linprog(...)
```

Description

`linprog` solves linear programming problems.

`x = linprog(f,A,b)` solves min `f'*x` such that `A*x <= b`.

`x = linprog(f,A,b,Aeq,beq)` solves the problem above while additionally satisfying the equality constraints `Aeq*x = beq`. Set `A=[]` and `b=[]` if no inequalities exist.

`x = linprog(f,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`. Set `Aeq=[]` and `beq=[]` if no equalities exist.

`x = linprog(f,A,b,Aeq,beq,lb,ub,x0)` sets the starting point to `x0`. This option is only available with the medium-scale algorithm (`options.LargeScale` is `'off'`). The default large-scale algorithm will ignore any starting point.

`x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`[x,fval] = linprog(...)` returns the value of the objective function `fun` at the solution `x`: `fval = f'*x`.

`[x,lambda,exitflag] = linprog(...)` returns a value `exitflag` that describes the exit condition.

`[x,lambda,exitflag,output] = linprog(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,exitflag,output,lambda] = linprog(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `linprog` are included below for `options`, `exitflag`, `lambda`, and `output`.

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function.

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output. At this time the `'iter'` level only works with the large-scale algorithm.
- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda`	A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type): • `lambda.lower` for the lower bounds `lb`. • `lambda.upper` for the upper bounds `ub`. • `lambda.ineqlin` for the linear inequalities. • `lambda.eqlin` for the linear equalities.
`output`	A structure whose fields contain information about the optimization: • `output.iterations` – The number of iterations taken. • `output.algorithm` – The algorithm used. • `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).

Examples

Find `x` that minimizes $f(x) = -5x_1 - 4x_2 - 6x_3$

subject to

$$\begin{aligned} x_1 - x_2 + x_3 &\le 20 \\ 3x_1 + 2x_2 + 4x_3 &\le 42 \\ 3x_1 + 2x_2 &\le 30 \\ 0 \le x_1, 0 \le x_2, 0 &\le x_3 \end{aligned}$$

First, enter the coefficients:

```
f = [-5; -4; -6]
A =  [1 -1  1
       3  2  4
       3  2  0];
b = [20; 42; 30];
lb = zeros(3,1);
```

Next, call a linear programming routine:

```
[x,fval,exitflag,output,lambda] = linprog(f,A,b,[],[],lb);
```

Entering `x`, `lambda.ineqlin`, and `lambda.lower` gets

```
x =
     0.0000
    15.0000
     3.0000
lambda.ineqlin =
     0
     1.5000
     0.5000
lambda.lower =
     1.0000
     0
     0
```

Nonzero elements of the vectors in the fields of `lambda` indicate active constraints at the solution. In this case, the second and third inequality constraints (in `lambda.ineqlin`) and the first lower bound constraint (in `lambda.lower`) are active constraints (i.e., the solution is on their constraint boundaries).

Algorithm

Large-scale optimization. The large-scale method is based on LIPSOL ([3]), which is a variant of Mehrotra's predictor-corrector algorithm ([2]), a primal-dual interior-point method. A number of preprocessing steps occur before the algorithm begins to iterate. See the linear programming section in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. `linprog` uses a projection method as used in the `quadprog` algorithm. `linprog` is an active set method and is thus a variation of the well-known *simplex* method for linear programming [1]. It finds an initial feasible solution by first solving another linear programming problem.

Diagnostics

Large-scale optimization. The first stage of the algorithm may involve some preprocessing of the constraints (see the "Large-Scale Linear Programming" section of the Large-Scale Algorithms chapter). Several possible conditions might occur that cause `linprog` to exit with an infeasibility message. In each case, the `exitflag` argument returned by `linprog` will be set to a negative value to indicate failure.

If a row of all zeros is detected in `Aeq` but the corresponding element of `beq` is not zero, the exit message is

```
Exiting due to infeasibility:  an all zero row in the constraint
matrix does not have a zero in corresponding right hand size
entry.
```

If one of the elements of `x` is found to not be bounded below, the exit message is

```
Exiting due to infeasibility:  objective f'*x is unbounded below.
```

If one of the rows of `Aeq` has only one nonzero element, the associated value in `x` is called a *singleton* variable. In this case, the value of that component of `x` can be computed from `Aeq` and `beq`. If the value computed violates another constraint, the exit message is

```
Exiting due to infeasibility: Singleton variables in equality
constraints are not feasible.
```

If the singleton variable can be solved for but the solution violates the upper or lower bounds, the exit message is

```
Exiting due to infeasibility: singleton variables in the equality
constraints are not within bounds.
```

Note: The preprocessing steps are cumulative. For example, even if your constraint matrix does not have a row of all zeros to begin with, other preprocessing steps may cause such a row to occur.

Once the preprocessing has finished, the iterative part algorithm begins until the stopping criteria is met. (See the “Large-Scale Linear Programming” section of the *Large-Scale Algorithms* chapter for more information about residuals, the primal problem, the dual problem, and the related stopping criteria.) If the residuals are growing instead of getting smaller, or the residuals are neither growing nor shrinking, one of the two following termination messages will display, respectively,

```
One or more of the residuals, duality gap, or total relative error
has grown 100000 times greater than its minimum value so far:
```

or

```
One or more of the residuals, duality gap, or total relative error
has stalled:
```

After one of these messages displays, it will be followed by one of the following six messages indicating if it appears the dual, the primal, or both are infeasible. The messages differ according to how the infeasibility or unboundedness was measured.

```
The dual appears to be infeasible (and the primal unbounded).(The
primal residual < TolFun)
```

```
The primal appears to be infeasible (and the dual unbounded). The
dual residual < TolFun)
```

```
The dual appears to be infeasible (and the primal unbounded) since
the dual residual > sqrt(TolFun).(The primal residual <
10*TolFun)
```

```
The primal appears to be infeasible (and the dual unbounded) since
the primal residual > sqrt(TolFun).(The dual residual <
10*TolFun)
```

```
The dual appears to be infeasible and the primal unbounded since
the primal objective < –1e+10 and the dual objective < 1e+6.
```

```
The primal appears to be infeasible and the dual unbounded since
the dual objective > 1e+10 and the primal objective > –1e+6.
```

```
Both the primal and the dual appear to be infeasible.
```

Note that, for example, the primal (objective) can be unbounded and the primal residual, which is a measure of primal constraint satisfaction, can be small.

Medium-scale optimization. `linprog` gives a warning when the solution is infeasible:

```
Warning: The constraints are overly stringent;
there is no feasible solution.
```

In this case, `linprog` produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, `linprog` gives

```
Warning: The equality constraints are overly
stringent; there is no feasible solution.
```

Unbounded solutions result in the warning

```
Warning: The solution is unbounded and at infinity;
the constraints are not restrictive enough
```

In this case, `linprog` returns a value of `x` that satisfies the constraints.

Limitations

Medium-scale optimization. At this time, the only levels of display, using the `Display` parameter in `options`, are `'off'` and `'final'`; iterative output using `'iter'` is not available.

See Also

`quadprog`

References

[1] Dantzig, G.B., A. Orden, and P. Wolfe, "Generalized Simplex Method for Minimizing a Linear from Under Linear Inequality Constraints," *Pacific Journal Math.* Vol. 5, pp. 183–195.

[2] Mehrotra, S., "On the Implementation of a Primal-Dual Interior Point Method," *SIAM Journal on Optimization*, Vol. 2, pp. 575-601, 1992.

[3] Zhang, Y., "Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment," *Technical Report TR96-01*, Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, MD, July 1995.

Purpose

Solve nonlinear curve-fitting (data-fitting) problems in the least-squares sense. That is, given input data *xdata,* and the observed output *ydata*, find coefficients *x* that "best-fit" the equation *F(x, xdata)*

$$\min_x \frac{1}{2}\|F(x,xdata)-ydata\|_2^2 = \frac{1}{2}\sum_i (F(x, xdata_i)-ydata_i)^2$$

where *xdata* and *ydata* are vectors and *F(x, xdata)* is a vector valued function.

The function `lsqcurvefit` uses the same algorithm as `lsqnonlin`. Its purpose is to provide an interface designed specifically for data-fitting problems.

Syntax

```
x = lsqcurvefit(fun,x0,xdata,ydata)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options,P1,P2,...)
[x,resnorm] = lsqcurvefit(...)
[x,resnorm,residual] = lsqcurvefit(...)
[x,resnorm,residual,exitflag] = lsqcurvefit(...)
[x,resnorm,residual,exitflag,output] = lsqcurvefit(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqcurvefit(...)
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
    lsqcurvefit(...)
```

Description

`lsqcurvefit` solves nonlinear data-fitting problems. `lsqcurvefit` requires a user-defined function to compute the vector-valued function *F(x, xdata)*. The size of the vector returned by the user-defined function must be the same as the size of *ydata*.

`x = lsqcurvefit(fun,x0,xdata,ydata)` starts at `x0` and finds coefficients `x` to best fit the nonlinear function `fun(x,xdata)` to the data `ydata` (in the least-squares sense). `ydata` must be the same size as the vector (or matrix) `F` returned by `fun`.

`x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Pass an empty matrix for `options` to use the default values for `options`.

`[x,resnorm] = lsqcurvefit(...)` returns the value of the squared 2-norm of the residual at `x`: `sum{(fun(x,xdata)–ydata).^2}`.

`[x,resnorm,residual] = lsqcurvefit(...)` returns the value of the residual, `fun(x,xdata)–ydata`, at the solution `x`.

`[x,resnorm,residual,exitflag] = lsqcurvefit(...)` returns a value `exitflag` that describes the exit condition.

`[x,resnorm,residual,exitflag,output] = lsqcurvefit(...)` returns a structure `output` that contains information about the optimization.

`[x,resnorm,residual,exitflag,output,lambda] = lsqcurvefit(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

`[x,resnorm,residual,exitflag,output,lambda,jacobian] = 99`
`99lsqcurvefit(...)` returns the Jacobian of `fun` at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to `lsqcurvefit` are included below for `fun`, `options`, `exitflag`, `lambda`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a vector `F` of the objective functions evaluated at `x`. You can specify `fun` to be an inline object with two input parameters `x` and `xdata`. For example,

```
f = ...
inline('x(1)*xdata.^2+x(2)*sin(xdata)','x','xdata');
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function F = myfun(x,xdata)
F = ...            % Compute function values at x
```

If the Jacobian can also be computed *and* `options.Jacobian` is `'on'`, set by

```
options = optimset('Jacobian','on')
```

then the function `fun` must return, in a second output argument, the Jacobian value `J`, a matrix, at `x`. Note that by checking the value of `nargout` the function can avoid computing `J` when `fun` is called with only one output argument (in the case where the optimization algorithm only needs the value of `F` but not `J`):

```
function [F,J] = myfun(x,xdata)
F = ...          % objective function values at x
if nargout > 1   % two output arguments
   J = ...   % Jacobian of the function evaluated at x
end
```

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, then the Jacobian `J` is an m-by-n matrix where `J(i,j)` is the partial derivative of `F(i)` with respect to `x(j)`. (Note that the Jacobian `J` is the transpose of the gradient of `F`.)

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale or medium-scale algorithm. For the large-scale algorithm, the nonlinear system of equations cannot be under-determined; that is, the number of equations (the number of elements of `F` returned by `fun`) must be at least as many as the length of `x`. Furthermore, only the large-scale algorithm handles bound constraints.

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `Jacobian` – Jacobian for the objective function defined by user. See the description of `fun` above to see how to define the Jacobian in `fun`.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `JacobPattern` – Sparsity pattern of the Jacobian for finite-differencing. If it is not convenient to compute the Jacobian matrix `J` in `fun`, `lsqcurvefit` can approximate `J` via sparse finite-differences provided the structure of `J`, i.e., locations of the nonzeros, is supplied as the value for `JacobPattern`. In the worst case, if the structure is unknown, you can set `JacobPattern` to be a dense matrix and a full finite-difference approximation will be computed in each iteration (this is the default if `JacobPattern` is not set). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

Parameters used by the medium-scale algorithm only:

- `DerivativeCheck` – Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
- `DiffMaxChange` – Maximum change in variables for finite-differencing.
- `DiffMinChange` – Minimum change in variables for finite-differencing.
- `LevenbergMarquardt` – Choose Levenberg-Marquardt over Gauss-Newton algorithm.
- `LineSearchType` – Line search algorithm choice.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`.
- `lambda.upper` for the upper bounds `ub`.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.stepsize` – The final step size taken (medium-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Note: The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See the examples below.

Examples

Vectors of data *xdata* and *ydata* are of length n. You want to find coefficients x to find the best fit to the equation

$$ydata(i) = x(1)\cdot xdata(i)^2 + x(2)\cdot \sin(xdata(i)) + x(3)\cdot xdata(i)^3$$

that is, you want to minimize

$$\min_{x} \ \frac{1}{2}\sum_{i=1}^{n} (F(x, xdata_i) - ydata_i)^2$$

where `F(x,xdata) = x(1)*xdata.^2 + x(2)*sin(xdata) + x(3)*xdata.^3`, starting at the point `x0 = [0.3, 0.4, 0.1]`.

First, write an M-file to return the value of `F` (`F` has `n` components):

```
function F = myfun(x,xdata)
F = x(1)*xdata.^2 + x(2)*sin(xdata) + x(3)*xdata.^3;
```

Next, invoke an optimization routine:

```
% Assume you determined xdata and ydata experimentally
xdata = [3.6 7.7 9.3 4.1 8.6 2.8 1.3 7.9 10.0 5.4];
ydata = [16.5 150.6 263.1 24.7 208.5 9.9 2.7 163.9 325.0 54.3];
x0 = [10, 10, 10]                    % Starting guess
[x,resnorm] = lsqcurvefit('myfun',x0,xdata,ydata)
```

Note that at the time that `lsqcurvefit` is called, `xdata` and `ydata` are assumed to exist and are vectors of the same size. They must be the same size because the value `F` returned by `fun` must be the same size as `ydata`.

After 33 function evaluations, this example gives the solution:

```
x =
0.2269    0.3385    0.3021
% residual or sum of squares
resnorm =
     6.2950
```

The residual is not zero because in this case there was some noise (experimental error) in the data.

Algorithm

Large-scale optimization. By default `lsqcurvefit` will choose the large-scale algorithm. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [5], [6]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. lsqcurvefit with options.LargeScale set to 'off' uses the Levenberg-Marquardt method with line-search [1], [2], [3]. Alternatively, a Gauss-Newton method [4] with line-search may be selected. The choice of algorithm is made by setting options.LevenbergMarquardt. Setting options.LevenbergMarquardt to 'off' (and options.LargeScale to 'off') selects the Gauss-Newton method, which is generally faster when the residual $\|F(x)\|_2^2$ is small.

The default line search algorithm, i.e., options.LineSearchType set to 'quadcubic', is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting options.LineSearchType to 'cubicpoly'. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. The algorithms used are described fully in the *Introduction to Algorithms* chapter.

Diagnostics

Large-scale optimization. The large-scale code will not allow equal upper and lower bounds. For example if lb(2)==ub(2) then lsqlin gives the error

```
Equal upper and lower bounds not permitted.
```

(lsqcurvefit does not handle equality constraints, which is another way to formulate equal bounds. If equality constraints are present, use fmincon, fminimax, or fgoalattain for alternative formulations where equality constraints can be included.)

Limitations

The function to be minimized must be continuous. lsqcurvefit may only give local solutions.

lsqcurvefit only handles real variables (the user-defined function must only return real values). When x has complex variables, the variables must be split into real and imaginary parts.

Large-scale optimization. The large-scale method for lsqcurvefit does not solve underdetermined systems: it requires that the number of equations, i.e., row dimension of F, be at least as great as the number of variables. In the underdetermined case, the medium-scale algorithm will be used instead. See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

The preconditioner computation used in the preconditioned conjugate gradient part of the large-scale method forms $J^T J$ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product $J^T J$, may lead to a costly solution process for large problems.

If components of x have no upper (or lower) bounds, then `lsqcurvefit` prefers that the corresponding components of `ub` (or `lb`) be set to `inf` (or `–inf` for lower bounds) as opposed to an arbitrary but very large positive (or negative for lower bounds) number.

Currently, if the analytical Jacobian is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic Jacobian to the finite-difference Jacobian. Instead, use the medium-scale method to check the derivatives with `options` parameter `MaxIter` set to zero iterations. Then run the problem with the large-scale method.

See Also

`optimset`, `lsqlin`, `lsqnonlin`, `lsqnonneg`, `\`

References

[1] Levenberg, K., “A Method for the Solution of Certain Problems in Least Squares,” *Quarterly Applied Math. 2*, pp. 164-168, 1944.

[2] Marquardt, D., “An Algorithm for Least–squares Estimation of Nonlinear Parameters,” *SIAM Journal Applied Math.* Vol. 11, pp. 431-441, 1963.

[3] More, J. J., “The Levenberg–Marquardt Algorithm: Implementation and Theory,” *Numerical Analysis*, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp. 105-116, 1977.

[4] Dennis, J. E. Jr., “Nonlinear Least Squares,” *State of the Art in Numerical Analysis*, ed. D. Jacobs, Academic Press, pp. 269-312, 1977.

[5] Coleman, T.F. and Y. Li, “On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds,” *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.

[6] Coleman, T.F. and Y. Li, “An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds,” *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

Purpose

Solve the constrained linear least-squares problem

$$\min_x \frac{1}{2}\|Cx - d\|_2^2 \qquad \text{such that} \qquad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

where C, A, and Aeq are matrices and d, b, beq, lb, ub, and x are vectors.

Syntax

```
x = lsqlin(C,d,A,b)
x = lsqlin(C,d,A,b,Aeq,beq)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)
[x,resnorm] = lsqlin(...)
[x,resnorm,residual] = lsqlin(...)
[x,resnorm,residual,exitflag] = lsqlin(...)
[x,resnorm,residual,exitflag,output] = lsqlin(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqlin(...)
```

Description

`x = lsqlin(C,d,A,b)` solves the linear system `C*x=d` in the least-squares sense subject to `A*x<=b`, where `C` is `m`-by-`n`.

`x = lsqlin(C,d,A,b,Aeq,beq)` solves the problem above while additionally satisfying the equality constraints `Aeq*x = beq`. Set `A=[]` and `b=[]` if no inequalities exist.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`. Set `Aeq=[]` and `beq=[]` if no equalities exist.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0)` sets the starting point to `x0`.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`[x,resnorm] = lsqlin(...)` returns the value of the squared 2-norm of the residual: `norm(C*x–d)^2`.

`[x,resnorm,residual] = lsqlin(...)` returns the residual, `C*x–d`.

`[x,resnorm,residual,exitflag] = lsqlin(...)` returns a value `exitflag` that describes the exit condition.

`[x,resnorm,residual,exitflag,output] = lsqlin(...)` returns a structure `output` that contains information about the optimization.

`[x,resnorm,residual,exitflag,output,lambda] = lsqlin(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `lsqlin` are included below for `options`, `exitflag`, `lambda`, and `output`.

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `lsqlin`, when the problem has *only* upper and lower bounds, i.e., no linear inequalities or equalities are specified, the default algorithm is the large-scale method. Otherwise the medium-scale algorithm will be used.

The parameter to set an algorithm preference:

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.

Parameters used by the large-scale algorithm only:

- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`.
- `lambda.upper` for the upper bounds `ub`.
- `lambda.ineqlin` for the linear inequalities.
- `lambda.eqlin` for the linear equalities.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Examples

Find the least-squares solution to the over-determined system $C \cdot x = d$ subject to $A \cdot x \leq b$ and $lb \leq x \leq ub$.

First, enter the coefficient matrices and the lower and upper bounds:

```
C = [
    0.9501    0.7620    0.6153    0.4057
    0.2311    0.4564    0.7919    0.9354
    0.6068    0.0185    0.9218    0.9169
    0.4859    0.8214    0.7382    0.4102
    0.8912    0.4447    0.1762    0.8936];
d = [
    0.0578
    0.3528
    0.8131
    0.0098
    0.1388];
A =[
    0.2027    0.2721    0.7467    0.4659
    0.1987    0.1988    0.4450    0.4186
    0.6037    0.0152    0.9318    0.8462];
b =[
    0.5251
    0.2026
    0.6721];
lb = –0.1*ones(4,1);
ub = 2*ones(4,1);
```

Next, call the constrained linear least-squares routine:

```
[x,resnorm,residual,exitflag,output,lambda] = ...
   lsqlin(C,d,A,b,[ ],[ ],lb,ub);
```

Entering `x`, `lambda.ineqlin`, `lambda.lower`, `lambda.upper` gets

```
x =
   –0.1000
   –0.1000
    0.2152
    0.3502
lambda.ineqlin =
         0
    0.2392
         0
lambda.lower =
    0.0409
    0.2784
         0
         0
lambda.upper =
         0
         0
         0
         0
```

Nonzero elements of the vectors in the fields of `lambda` indicate active constraints at the solution. In this case, the second inequality constraint (in `lambda.ineqlin`) and the first lower and second lower bound constraints (in `lambda.lower`) are active constraints (i.e., the solution is on their constraint boundaries).

Notes

For problems with no constraints, `\` should be used: `x= A\b`.

In general `lsqlin` locates a local solution.

Better numerical results are likely if you specify equalities explicitly using `Aeq` and `beq`, instead of implicitly using `lb` and `ub`.

Large-scale optimization. If `x0` is not strictly feasible, `lsqlin` chooses a new strictly feasible (centered) starting point.

If components of x have no upper (or lower) bounds, then `lsqlin` prefers that the corresponding components of `ub` (or `lb`) be set to `Inf` (or `–Inf` for `lb`) as opposed to an arbitrary but very large positive (or negative in the case of lower bounds) number.

Algorithm

Large-scale optimization. When the problem given to lsqlin has *only* upper and lower bounds, i.e., no linear inequalities or equalities are specified, and the matrix `C` has at least as many rows as columns, the default algorithm is the large-scale method. This method is a subspace trust-region method based on the interior-reflective Newton method described in [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. `lsqlin` with `options.LargeScale` set to `'off'`, or when linear inequalities or equalities are given, is based on `quadprog`, which uses an active set method similar to that described in [1]. It finds an initial feasible solution by first solving a linear programming problem. See the quadratic programming method discussed in the *Introduction to Algorithms* chapter.

Diagnostics

Large-scale optimization. The large-scale code does not allow equal upper and lower bounds. For example if `lb(2)==ub(2)`, then `lsqlin` gives the error:

```
Equal upper and lower bounds not permitted in this large-scale
method.
Use equality constraints and the medium-scale method instead.
```

At this time, the medium-scale algorithm must be used to solve equality constrained problems.

Medium-scale optimization. If the matrices `C`, `A` or `Aeq` are sparse, and the problem formulation is not solvable using the large-scale code, `lsqlin` warns that the matrices will be converted to full:

```
Warning: This problem formulation not yet available for sparse
matrices.
Converting to full to solve.
```

lsqlin gives a warning when the solution is infeasible:

```
Warning: The constraints are overly stringent;
     there is no feasible solution.
```

In this case, lsqlin produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, lsqlin gives

```
Warning: The equality constraints are overly stringent;
     there is no feasible solution.
```

Limitations

At this time, the only levels of display, using the Display parameter in options, are 'off' and 'final'; iterative output using 'iter' is not available.

See Also

lsqnonneg, quadprog, \

References

[1] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization,* Academic Press, London, UK, 1981.

[2] Coleman, T.F. and Y. Li, "A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on Some of the Variables", *SIAM Journal on Optimization*, Vol. 6, Number 4, pp. 1040-1058, 1996.

lsqnonlin

Purpose

Solve nonlinear least-squares (nonlinear data-fitting) problems

$$\min_x f(x) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + \ldots + f_m(x)^2 + L$$

where L is a constant.

Syntax

```
x = lsqnonlin(fun,x0)
x = lsqnonlin(fun,x0,lb,ub)
x = lsqnonlin(fun,x0,lb,ub,options)
x = lsqnonlin(fun,x0,options,P1,P2, ... )
[x,resnorm] = lsqnonlin(...)
[x,resnorm,residual] = lsqnonlin(...)
[x,resnorm,residual,exitflag] = lsqnonlin(...)
[x,resnorm,residual,exitflag,output] = lsqnonlin(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonlin(...)
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
    lsqnonlin(...)
```

Description

`lsqnonlin` solves nonlinear least-squares problems, including nonlinear data-fitting problems.

Rather than compute the value *f(x)* (the “sum of squares”), `lsqnonlin` requires the user-defined function to compute the *vector*-valued function

$$F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix}$$

Then, in vector terms, this optimization problem may be restated as

$$\min_x \frac{1}{2}\|F(x)\|_2^2 = \frac{1}{2}\sum_i f_i(x)^2$$

where x is a vector and $F(x)$ is a function that returns a vector value.

`x = lsqnonlin(fun,x0)` starts at the point `x0` and finds a minimum to the sum of squares of the functions described in `fun`. `fun` should return a vector of values and not the sum-of-squares of the values. (`fun(x)` is summed and squared implicitly in the algorithm.)

`x = lsqnonlin(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = lsqnonlin(fun,x0,lb,ub,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = lsqnonlin(fun,x0,lb,ub,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Pass an empty matrix for `options` to use the default values for `options`.

`[x,resnorm] = lsqnonlin(...)` returns the value of the squared 2-norm of the residual at `x`: `sum(fun(x).^2)`.

`[x,resnorm,residual] = lsqnonlin(...)` returns the value of the residual, `fun(x)`, at the solution `x`.

`[x,resnorm,residual,exitflag] = lsqnonlin(...)` returns a value `exitflag` that describes the exit condition.

`[x,resnorm,residual,exitflag,output] = lsqnonlin(...)` returns a structure `output` that contains information about the optimization.

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonlin(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

`[x,resnorm,residual,exitflag,output,lambda,jacobian] = lsqnonlin(...)` returns the Jacobian of `fun` at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details

relevant to `lsqnonlin` are included below for `fun`, `options`, `exitflag`, `lambda`, and `output`.

`fun` The function to be minimized. `fun` takes a vector `x` and returns a vector `F` of the objective functions evaluated at `x`. You can specify `fun` to be an inline object. For example,

```
x = lsqnonlin(inline('sin(x.*x)'),x0)
```

Alternatively, `fun` can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If `fun='myfun'` then the M-file function `myfun.m` would have the form

```
function F = myfun(x)
F = ...             % Compute function values at x
```

If the Jacobian can also be computed *and* `options.Jacobian` is `'on'`, set by

```
options = optimset('Jacobian','on')
```

then the function `fun` must return, in a second output argument, the Jacobian value `J`, a matrix, at `x`. Note that by checking the value of `nargout` the function can avoid computing `J` when `fun` is called with only one output argument (in the case where the optimization algorithm only needs the value of `F` but not `J`):

```
function [F,J] = myfun(x)
F = ...           % objective function values at x
if nargout > 1    % two output arguments
   J = ...   % Jacobian of the function evaluated at x
end
```

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, then the Jacobian `J` is an m-by-n matrix where `J(i,j)` is the partial derivative of `F(i)` with respect to `x(j)`. (Note that the Jacobian `J` is the transpose of the gradient of `F`.)

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference because certain conditions must be met to use the large-scale or medium-scale algorithm. For the large-scale algorithm, the nonlinear system of equations cannot be under-determined; that is, the number of equations (the number of elements of `F` returned by `fun`) must be at least as many as the length of `x`. Furthermore, only the large-scale algorithm handles bound constraints.

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `Jacobian` – Jacobian for the objective function defined by user. See the description of `fun` above to see how to define the Jacobian in `fun`.
- `MaxFunEvals` – Maximum number of function evaluations allowed.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `JacobPattern` – Sparsity pattern of the Jacobian for finite-differencing. If it is not convenient to compute the Jacobian matrix `J` in `fun`, `lsqnonlin` can approximate `J` via sparse finite-differences provided the structure of `J`, i.e., locations of the nonzeros, is supplied as the value for `JacobPattern`. In the worst case, if the structure is unknown, you can set `JacobPattern` to be a dense matrix and a full finite-difference approximation will be computed in each iteration (this is the default if `JacobPattern` is not set). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

Parameters used by the medium-scale algorithm only:

- `DerivativeCheck` – Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
- `DiffMaxChange` – Maximum change in variables for finite-differencing.
- `DiffMinChange` – Minimum change in variables for finite-differencing.
- `LevenbergMarquardt` – Choose Levenberg-Marquardt over Gauss-Newton algorithm.
- `LineSearchType` – Line search algorithm choice.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`.
- `lambda.upper` for the upper bounds `ub`.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.funcCount` – The number of function evaluations.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.stepsize` – The final step size taken (medium-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Note: The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See the example below.

Examples

Find x that minimizes

$$\sum_{k=1}^{10} (2 + 2k - e^{kx_1} - e^{kx_2})^2$$

starting at the point `x = [0.3, 0.4]`.

Because lsqnonlin assumes that the sum-of-squares is *not* explicitly formed in the user function, the function passed to lsqnonlin should instead compute the vector valued function

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2}$$

for $k = 1$ to 10 (that is, F should have k components).

First, write an M-file to compute the k-component vector F:

```
function F = myfun(x)
k = 1:10;
F = 2 + 2*k–exp(k*x(1))–exp(k*x(2));
```

Next, invoke an optimization routine:

```
x0 = [0.3 0.4]            % Starting guess
[x,resnorm] = lsqnonlin('myfun',x0)     % Invoke optimizer
```

After about 24 function evaluations, this example gives the solution:

```
x =
     0.2578   0.2578
resnorm     %residual or sum of squares
resnorm =
     124.3622
```

Algorithm

Large-scale optimization. By default lsqnonlin will choose the large-scale algorithm. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [5], [6]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. lsqnonlin with options.LargeScale set to 'off' uses the Levenberg-Marquardt method with line-search [1], [2], [3]. Alternatively, a Gauss-Newton method [4] with line-search may be selected. The choice of algorithm is made by setting options.LevenbergMarquardt. Setting options.LevenbergMarquardt to 'off' (and options.LargeScale to 'off') selects the Gauss-Newton method, which is generally faster when the residual $\|F(x)\|_2^2$ is small.

The default line search algorithm, i.e., `options.LineSearchType` set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options.LineSearchType` to `'cubicpoly'`. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. The algorithms used are described fully in the *Introduction to Algorithms* chapter.

Diagnostics

Large-scale optimization. The large-scale code will not allow equal upper and lower bounds. For example if `lb(2)==ub(2)` then `lsqlin` gives the error:

```
Equal upper and lower bounds not permitted.
```

(`lsqnonlin` does not handle equality constraints, which is another way to formulate equal bounds. If equality constraints are present, use `fmincon`, `fminimax` or `fgoalattain` for alternative formulations where equality constraints can be included.)

Limitations

The function to be minimized must be continuous. `lsqnonlin` may only give local solutions.

`lsqnonlin` only handles real variables. When x has complex variables, the variables must be split into real and imaginary parts.

Large-scale optimization. The large-scale method for `lsqnonlin` does not solve under-determined systems: it requires that the number of equations (i.e., the number of elements of F) be at least as great as the number of variables. In the under-determined case, the medium-scale algorithm will be used instead. (If bound constraints exist, a warning will be issued and the problem will be solved with the bounds ignored.) See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

The preconditioner computation used in the preconditioned conjugate gradient part of the large-scale method forms J^TJ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product J^TJ, may lead to a costly solution process for large problems.

If components of x have no upper (or lower) bounds, then `lsqnonlin` prefers that the corresponding components of `ub` (or `lb`) be set to `inf` (or `–inf` for lower bounds) as opposed to an arbitrary but very large positive (or negative for lower bounds) number.

Currently, if the analytical Jacobian is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic Jacobian to the finite-difference Jacobian. Instead, use the medium-scale method to check the derivatives with `options` parameter `MaxIter` set to 0 iterations. Then run the problem with the large-scale method.

Medium-scale optimization. The medium-scale algorithm does not handle bound constraints.

Since the large-scale algorithm does not handle under-determined systems and the medium-scale does not handle bound constraints, problems with both these characteristics cannot be solved by `lsqnonlin`.

See Also

`optimset`, `lsqcurvefit`, `lsqlin`

References

[1] Levenberg, K.,"A Method for the Solution of Certain Problems in Least Squares," *Quarterly Applied Math. 2*, pp. 164-168, 1944.

[2] Marquardt, D.,"An Algorithm for Least–squares Estimation of Nonlinear Parameters," *SIAM J. Applied Math.* Vol. 11, pp. 431-441, 1963.

[3] Moré, J.J., "The Levenberg–Marquardt Algorithm: Implementation and Theory," *Numerical Analysis*, ed. G. A. Watson, *Lecture Notes in Mathematics* 630, Springer Verlag, pp. 105-116, 1977.

[4] Dennis, J.E., Jr., "Nonlinear Least Squares," *State of the Art in Numerical Analysis*, ed. D. Jacobs, Academic Press, pp. 269-312, 1977.

[5] Coleman, T.F. and Y. Li, "On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds," Mathematical Programming, Vol. 67, Number 2, pp. 189-224, 1994.

[6] Coleman, T.F. and Y. Li, "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds," *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

Purpose

Solves the nonnegative least squares problem

$$\min_{x} \frac{1}{2}\|Cx - d\|_2^2 \qquad \text{such that} \qquad x \geq 0$$

where the matrix C and the vector d are the coefficients of the objective function. The vector, x, of independent variables is restricted to be nonnegative.

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,x0)
x = lsqnonneg(C,d,x0,options)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

Description

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x–d)` subject to `x >= 0`. `C` and `d` must be real.

`x = lsqnonneg(C,d,x0)` uses `x0` as the starting point if all `x0 >= 0`; otherwise, the default is used. The default start point is the origin (the default is also used when `x0==[]` or when only two input arguments are provided).

`x = lsqnonneg(C,d,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual: `norm(C*x–d)^2`.

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual, `C*x–d`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the optimization.

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the Lagrange multipliers in the vector `lambda`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `lsqnonneg` are included below for `options`, `exitflag`, `lambda`, and `output`.

`options`	Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. • `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output. • `TolX` – Termination tolerance on `x`.
`exitflag`	Describes the exit condition: • `> 0` indicates that the function converged to a solution `x`. • 0 indicates the iteration count was exceeded. Increasing the tolerance `TolX` may lead to a solution.
`lambda`	Vector containing the Lagrange multipliers: `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.
`output`	A structure whose fields contain information about the optimization: • `output.iterations` – The number of iterations taken. • `output.algorithm` – The algorithm used.

Examples

Compare the unconstrained least squares solution to the lsqnonneg solution for a 4-by-2 problem.

```
C = [
     0.0372    0.2869
     0.6861    0.7071
     0.6233    0.6245
     0.6344    0.6170];

d = [
     0.8587
     0.1781
     0.0747
     0.8405];

[C\d, lsqnonneg(C,d)] =
    –2.5627    0
     3.1108    0.6929

[norm(C*(C\d)–d), norm(C*lsqnonneg(C,d)–d)] =
        0.6674 0.9118
```

The solution from lsqnonneg does not fit as well as the least squares solution. However, the nonnegative least-squares solution has no negative components.

Algorithm

lsqnonneg uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector lambda. It then selects the basis vector corresponding to the maximum value in lambda in order to swap it out of the basis in exchange for another possible candidate. This continues until lambda <= 0.

Notes

The nonnegative least squares problem is a subset of the constrained linear least-squares problem. Thus, when C has more rows than columns (i.e., the system is over-determined)

```
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(C,d)
```

is equivalent to

```
[m,n] = size(C);
[x,resnorm,residual,exitflag,output,lambda_lsqlin] =
   lsqlin(C,d,—eye(n,n),zeros(n,1));
```

except that `lambda = —lambda_lsqlin.ineqlin`.

For problems greater than order twenty, `lsqlin` may be faster than `lsqnonneg`, otherwise `lsqnonneg` is generally more efficient.

See Also `optimset`, `lsqlin`, `\`

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Chapter 23, p. 161, 1974.

Purpose

Get optimization options parameter values.

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`:

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`:

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset`

Purpose

Create or edit optimization options parameter structure.

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options parameter structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options` is passed to the optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(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`.

Parameters

For more information about individual parameters, see the reference pages for the optimization functions that use these parameters, or Table 4-3.

In the lists below, values in { } denote the default value; some parameters have different defaults for different optimization functions and so no values are shown in { }.

Optimization parameters used by both large-scale and medium-scale algorithms:

Diagnostics	[on | {off}]
Display	[off | iter | {final}]
GradObj	[on | {off}]
Jacobian	[on | {off}]
LargeScale	[{on} | off]
MaxFunEvals	[positive integer]
MaxIter	[positive integer]
TolCon	[positive scalar]
TolFun	[positive scalar]
TolX	[positive scalar]

Optimization parameters used by large-scale algorithms only:

Hessian	[on | {off}]
HessPattern	[sparse matrix]
JacobPattern	[sparse matrix]
MaxPCGIter	[positive integer]
PrecondBandWidth	[positive integer | Inf]
TolPCG	[positive scalar | {0.1}]
TypicalX	[vector]

Optimization parameters used by medium-scale algorithms only:

```
DerivativeCheck     [ on | {off} ]
DiffMaxChange       [ positive scalar | {1e–1} ]
DiffMinChange       [ positive scalar | {1e–8} ]
GoalsExactAchieve   [ positive scalar integer | {0} ]
GradConstr          [ on | {off} ]
HessUpdate          [ {bfgs} | dfp | gillmurray | steepdesc ]
LevenbergMarquardt  [ on | off ]
LineSearchType      [ cubicpoly | {quadcubic} ]
MeritFunction       [ singleobj | {multiobj} ]
MinAbsMax           [ positive scalar integer | {0} ]
```

Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to `'iter'` and the `TolFun` parameter is set to `1e–8`:

```
options = optimset('Display','iter','TolFun',1e–8)
```

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`:

```
optnew = optimset(options,'TolX',1e–4);
```

This statement returns an optimization options structure `options` that contains all the parameter names and default values relevant to the function `fminbnd`:

```
options = optimset('fminbnd')
```

If you only want to see the default values for `fminbnd`, you can simply type

```
optimset fminbnd
```

or equivalently

```
optimset('fminbnd')
```

See Also

`optimget`

Purpose

Solve the quadratic programming problem

$$\min_{x} \frac{1}{2}x^T H x + f^T x \quad \text{such that} \quad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

where H, A, and Aeq are matrices, and f, b, beq, lb, ub, and x are vectors.

Syntax

```
x = quadprog(H,f,A,b)
x = quadprog(H,f,A,b,Aeq,beq)
x = quadprog(H,f,A,b,Aeq,beq,lb,ub)
x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0)
x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options)
[x,fval] = quadprog(...)
[x,fval,exitflag] = quadprog(...)
[x,fval,exitflag,output] = quadprog(...)
[x,fval,exitflag,output,lambda] = quadprog(...)
```

Description

`x = quadprog(H,f,A,b)` returns a vector `x` that minimizes `1/2*x'*H*x + f'*x` subject to `A*x <= b`.

`x = quadprog(H,f,A,b,Aeq,beq)` solves the problem above while additionally satisfying the equality constraints `Aeq*x = beq`.

`x = quadprog(H,f,A,b,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is in the range `lb <= x <= ub`.

`x = quadprog(H,f,A,b,lb,ub,x0)` sets the starting point to `x0`.

`x = quadprog(H,f,A,b,lb,ub,x0,options)` minimizes with the optimization parameters specified in the structure `options`.

`[x,fval] = quadprog(...)` returns the value of the objective function at x: `fval = 0.5*x'*H*x + f'*x`.

`[x,fval,exitflag] = quadprog(...)` returns a value `exitflag` that describes the exit condition of `quadprog`.

`[x,fval,exitflag,output] = quadprog(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,exitflag,output,lambda] = quadprog(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Arguments

The arguments passed into the function are described in Table 4-1. The arguments returned by the function are described in Table 4-2. Details relevant to `quadprog` are included below for `options`, `exitflag`, `lambda`, and `output`.

`options` Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `quadprog`, when the problem has *only* upper and lower bounds, i.e., no linear inequalities or equalities are specified, the default algorithm is the large-scale method. Or, if the problem given to `quadprog` has *only* linear equalities, i.e., no upper and lower bounds or linear inequalities are specified, and the number of equalities is no greater than the length of `x`, the default algorithm is the large-scale method. Otherwise the medium-scale algorithm will be used.

The parameter to set an algorithm preference:

- `LargeScale` – Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Parameters used by both the large-scale and medium-scale algorithms:

- `Diagnostics` – Print diagnostic information about the function to be minimized.
- `Display` – Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output.
- `MaxIter` – Maximum number of iterations allowed.
- `TolFun` – Termination tolerance on the function value.
- `TolX` – Termination tolerance on `x`.

Parameters used by the large-scale algorithm only:

- `MaxPCGIter` – Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below).
- `PrecondBandWidth` – Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
- `TolPCG` – Termination tolerance on the PCG iteration.
- `TypicalX` – Typical `x` values.

`exitflag` Describes the exit condition:

- `> 0` indicates that the function converged to a solution `x`.
- `0` indicates that the maximum number of function evaluations or iterations was reached.
- `< 0` indicates that the function did not converge to a solution.

`lambda` A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type):

- `lambda.lower` for the lower bounds `lb`.
- `lambda.upper` for the upper bounds `ub`.
- `lambda.ineqlin` for the linear inequalities.
- `lambda.eqlin` for the linear equalities.

`output` A structure whose fields contain information about the optimization:

- `output.iterations` – The number of iterations taken.
- `output.algorithm` – The algorithm used.
- `output.cgiterations` – The number of PCG iterations (large-scale algorithm only).
- `output.firstorderopt` – A measure of first-order optimality (large-scale algorithm only).

Examples

Find values of `x` that minimize

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2$$

subject to

$$x_1 + x_2 \le 2$$
$$-x_1 + 2x_2 \le 2$$
$$2x_1 + x_2 \le 3$$
$$0 \le x_1, 0 \le x_2$$

First note that this function may be written in matrix notation as $f(x) = \frac{1}{2}x^T Hx + f^T x$ where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \qquad f = \begin{bmatrix} -2 \\ -6 \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Enter these coefficient matrices:

```
H = [1 -1; -1 2]
f = [-2; -6]
A = [1 1; -1 2; 2 1]
b = [2; 2; 3]
lb = zeros(2,1)
```

Next, invoke a quadratic programming routine:

```
[x,fval,exitflag,output,lambda] = quadprog(H,f,A,b,[],[],lb)
```

This generates the solution

```
x =
     0.6667
     1.3333
fval =
    –8.2222
exitflag =
     1
output =
       iterations: 3
        algorithm: 'medium-scale: active-set'
    firstorderopt: []
     cgiterations: []
lambda.ineqlin
ans =
    3.1111
    0.4444
         0
lambda.lower
ans =
     0
     0
```

Nonzero elements of the vectors in the fields of `lambda` indicate active constraints at the solution. In this case, the first and second inequality constraints (in `lambda.ineqlin`) are active constraints (i.e., the solution is on their constraint boundaries). For this problem, all the lower bounds are inactive.

Notes

In general `quadprog` locates a local solution unless the problem is strictly convex.

Better numerical results are likely if you specify equalities explicitly using `Aeq` and `beq`, instead of implicitly using `lb` and `ub`.

If components of x have no upper (or lower) bounds, then `quadprog` prefers that the corresponding components of `ub` (or `lb`) be set to `Inf` (or `–Inf` for `lb`) as opposed to an arbitrary but very large positive (or negative in the case of lower bounds) number.

Large-scale optimization. If you do not supply x0, or x0 is not strictly feasible, quadprog chooses a new strictly feasible (centered) starting point.

If an equality constrained problem is posed and quadprog detects negative curvature, then the optimization terminates because the constraints are not restrictive enough. In this case, exitflag is returned with the value –1, a message is displayed (unless the options Display parameter is 'off'), and the x returned is not a solution but a direction of negative curvature with respect to H.

Algorithm

Large-scale optimization. When the problem given to quadprog has *only* upper and lower bounds, i.e., no linear inequalities or equalities are specified, the default algorithm is the large-scale method. Or, if the problem given to quadprog has *only* linear equalities, i.e., no upper and lower bounds or linear inequalities are specified the default algorithm is the large-scale method.

This method is a subspace trust-region method based on the interior-reflective Newton method described in [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

Medium-scale optimization. quadprog uses an active set method, which is also a projection method, similar to that described in [1]. It finds an initial feasible solution by first solving a linear programming problem. This method is discussed in the *Introduction to Algorithms* chapter.

Diagnostics

Large-scale optimization. The large-scale code will not allow equal upper and lower bounds. For example if lb(2)==ub(2) then quadprog gives the error:

```
Equal upper and lower bounds not permitted in this large-scale
method.
Use equality constraints and the medium-scale method instead.
```

If you only have equality constraints you can still use the large-scale method. But if you have both equalities and bounds, you must use the medium-scale method.

Medium-scale optimization. When the solution is infeasible, `quadprog` gives this warning:

```
Warning: The constraints are overly stringent;
     there is no feasible solution.
```

In this case, `quadprog` produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, `quadprog` gives this warning:

```
Warning: The equality constraints are overly stringent;
     there is no feasible solution.
```

Unbounded solutions,which can occur when the Hessian `H` is negative semidefinite, may result in

```
Warning: The solution is unbounded and at infinity;
     the constraints are not restrictive enough.
```

In this case, `quadprog` returns a value of `x` that satisfies the constraints.

Limitations

At this time the only levels of display, using the `Display` parameter in `options`, are `'off'` and `'final'`; iterative output using `'iter'` is not available.

The solution to indefinite or negative definite problems is often unbounded (in this case, `exitflag` is returned with a negative value to show a minimum was not found); when a finite solution does exist, `quadprog` may only give local minima since the problem may be nonconvex.

Large-scale optimization. The linear equalities cannot be dependent (i.e., `Aeq` must have full row rank). Note that this means that `Aeq` cannot have more rows than columns. If either of these cases occur, the medium-scale algorithm will be called instead. See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

References

[1] P.E. Gill, W. Murray, and M.H. Wright, *Practical Optimization,* Academic Press, London, UK, 1981.

[2] Coleman, T.F. and Y. Li, "A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on some of the Variables," *SIAM Journal on Optimization*, Vol. 6, Number 4, pp. 1040-1058, 1996.

Index