Optimization Toolbox

For Use with MATLAB®

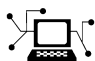
- Computation
- Visualization
- Programming

User's Guide

Version 3

The MathWorks

How to Contact The MathWorks:



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

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

508-647-7000 Phone

508-647-7001 Fax

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

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

Optimization Toolbox User's Guide

© COPYRIGHT 1990 - 2004 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:			
	November 1990	First printing	
	December 1996	Second printing	For MATLAB 5
	January 1999	Third printing	For Version 2 (Release 11)
	September 2000	Fourth printing	For Version 2.1 (Release 12)
	June 2001	Online only	Revised for Version 2.1.1 (Release 12.1)
	September 2003	Online only	Revised for Version 2.3 (Release 13SP1)
	June 2004	Fifth printing	Revised for Version 3.0 (Release 14)

Acknowledgements

The MathWorks would like to acknowledge the following contributors to the Optimization Toolbox.

Thomas F. Coleman researched and contributed the large-scale algorithms for constrained and unconstrained minimization, nonlinear least squares and curve fitting, constrained linear least squares, quadratic programming, and nonlinear equations.

Dr. Coleman is Professor of Computer Science and Applied Mathematics at Cornell University. He is Director of the Cornell Theory Center and the Cornell Computational Finance Institute. Dr. Coleman is Chair of the SIAM Activity Group on Optimization, and a member of the Editorial Boards of *Applied Mathematics Letters*, *SIAM Journal of Scientific Computing*, *Computational Optimization and Applications*, *Communications on Applied Nonlinear Analysis*, and *Mathematical Modeling and Scientific Computing*.

Dr. Coleman has published 4 books and over 70 technical papers in the areas of continuous optimization and computational methods and tools for large-scale problems.

Yin Zhang researched and contributed the large-scale linear programming algorithm.

Dr. Zhang is Associate Professor of Computational and Applied Mathematics on the faculty of the Keck Center for Computational Biology at Rice University. He is on the Editorial Board of *SIAM Journal on Optimization*, and is Associate Editor of *Journal of Optimization: Theory and Applications*.

Dr. Zhang has published over 40 technical papers in the areas of interior-point methods for linear programming and computation mathematical programming.

Contents

1 Getting Started

What Is the Optimization Toolbox? **1-2**

Optimization Example **1-3**
The Problem **1-3**
Setting Up the Problem **1-3**
Finding the Solution **1-4**
More Examples **1-5**

2 Tutorial

Introduction **2-3**
Problems Covered by the Toolbox **2-3**
Using the Optimization Functions **2-6**
Medium- and Large-Scale Algorithms **2-7**

Examples That Use Standard Algorithms **2-9**
Unconstrained Minimization Example **2-10**
Nonlinear Inequality Constrained Example **2-11**
Constrained Example with Bounds **2-13**
Constrained Example with Gradients **2-14**
Gradient Check: Analytic Versus Numeric **2-16**
Equality Constrained Example **2-17**
Maximization **2-18**
Greater-Than-Zero Constraints **2-18**
Avoiding Global Variables via Anonymous and Nested Functions **2-19**
Nonlinear Equations with Analytic Jacobian **2-23**
Nonlinear Equations with Finite-Difference Jacobian **2-26**
Multiobjective Examples **2-27**

Large-Scale Examples **2-40**
Problems Covered by Large-Scale Methods **2-41**
Nonlinear Equations with Jacobian **2-44**
Nonlinear Equations with Jacobian Sparsity Pattern **2-47**
Nonlinear Least-Squares with Full Jacobian Sparsity Pattern **2-49**
Nonlinear Minimization with Gradient and Hessian **2-51**
Nonlinear Minimization with Gradient and Hessian Sparsity Pattern **2-52**
Nonlinear Minimization with Bound Constraints and Banded Preconditioner **2-54**
Nonlinear Minimization with Equality Constraints **2-58**
Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints **2-59**
Quadratic Minimization with Bound Constraints **2-63**
Quadratic Minimization with a Dense but Structured Hessian **2-65**
Linear Least-Squares with Bound Constraints **2-70**
Linear Programming with Equalities and Inequalities **2-72**
Linear Programming with Dense Columns in the Equalities . **2-73**

Default Options Settings **2-76**
Changing the Default Settings **2-76**

Displaying Iterative Output **2-79**
Output Headings: Medium-Scale Algorithms **2-79**
Output Headings: Large-Scale Algorithms **2-82**

Calling an Output Function Iteratively **2-85**
Creating the Output Function **2-85**
Running the Example **2-87**

Optimizing Anonymous Functions Instead of M-Files **2-90**

Typical Problems and How to Deal with Them **2-92**

Selected Bibliography **2-96**

3 Standard Algorithms

Optimization Overview **3-3**

Unconstrained Optimization **3-4**
Quasi-Newton Methods **3-6**
Line Search **3-8**

Quasi-Newton Implementation **3-10**
Hessian Update **3-10**
Line Search Procedures **3-10**

Least-Squares Optimization **3-17**
Gauss-Newton Method **3-18**
Levenberg-Marquardt Method **3-19**
Nonlinear Least-Squares Implementation **3-21**

Nonlinear Systems of Equations **3-23**
Gauss-Newton Method **3-23**
Trust-Region Dogleg Method **3-23**
Nonlinear Equations Implementation **3-25**

Constrained Optimization **3-27**
Sequential Quadratic Programming (SQP) **3-28**
Quadratic Programming (QP) Subproblem **3-29**
SQP Implementation **3-30**
Simplex Algorithm **3-36**

Multiobjective Optimization **3-41**
Introduction **3-41**
Goal Attainment Method **3-47**
Algorithm Improvements for Goal Attainment Method **3-48**

Selected Bibliography **3-51**

4 Large-Scale Algorithms

Trust-Region Methods for Nonlinear Minimization 4-2

Preconditioned Conjugate Gradients 4-5

Linearly Constrained Problems 4-7
Linear Equality Constraints 4-7
Box Constraints 4-7

Nonlinear Least-Squares 4-10

Quadratic Programming 4-11

Linear Least-Squares 4-12

Large-Scale Linear Programming 4-13
Main Algorithm 4-13
Preprocessing 4-16

Selected Bibliography 4-17

5 Function Reference

Functions — Categorical List 5-2
Minimization 5-2
Equation Solving 5-2
Least Squares (Curve Fitting) 5-3
Utility 5-3
Demos of Large-Scale Methods 5-3
Demos of Medium-Scale Methods 5-4

Function Arguments 5-5
Input Arguments 5-5
Output Arguments 5-7

Optimization Options **5-9**
Output Function **5-15**

Functions — Alphabetical List **5-25**

Index

1

Getting Started

What Is the Optimization Toolbox? (p. 1-2)	Introduces the toolbox and describes the types of problems it is designed to solve.
Optimization Example (p. 1-3)	Presents an example that illustrates how to use the toolbox.

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
- Sparse and structured large-scale problems

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

```
type function_name
```

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

Optimization Example

This section presents an example that illustrates how to solve an optimization problem using the toolbox function `lsqlin`, which solves linear least squares problems. This section covers the following topics:

- “The Problem” on page 1-3
- “Setting Up the Problem” on page 1-3
- “Finding the Solution” on page 1-4
- “More Examples” on page 1-5

The Problem

The problem in this example is to find the point on the plane $x_1 + 2x_2 + 4x_3 = 7$ that is closest to the origin. The easiest way to solve this problem is to minimize the square of the distance from a point $x = (x_1, x_2, x_3)$ on the plane to the origin, which returns the same optimal point as minimizing the actual distance. Since the square of the distance from an arbitrary point (x_1, x_2, x_3) to the origin is $x_1^2 + x_2^2 + x_3^2$, you can describe the problem as follows:

$$\underset{x}{\text{minimize}}\ f(x) = x_1^2 + x_2^2 + x_3^2$$

subject to the constraint

$$x_1 + 2x_2 + 4x_3 = 7$$

The function *f*(*x*) is called the *objective function* and $x_1 + 2x_2 + 4x_3 = 7$ is an *equality constraint*. More complicated problems might contain other equality constraints, inequality constraints, and upper or lower bound constraints.

Setting Up the Problem

This section shows how to set up the problem before applying the function `lsqlin`, which solves linear least squares problems of the form

$$\underset{x}{\text{minimize}}\ f(x) = \|Cx - d\|^2$$

where $\|Cx - d\|^2$ is the norm of *Cx* - *d* squared, subject to the constraints

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

To set up the problem, you must create variables for the parameters *C*, *d*, *A*, *b*, *Aeq*, and *beq*. `lsqlin` accepts these variables as input arguments with the following syntax:

```
x = lsqlin(C, d, A, b, Aeq, beq)
```

To create the variables, do the following steps:

1. Create Variables for the Objective Function

Since you want to minimize $x_1^2 + x_2^2 + x_3^2 = \|x\|^2$, you can set *C* to be the 3-by-3 identity matrix and *d* to be a 3-by-1 vector of zeros, so that *Cx* - *d* = *x*.

```
C = eye(3);
d = zeros(3,1);
```

2. Create Variables for the Constraints

Since this examples has no inequality constraints, you can set `A` and `b` to be empty matrices in the input arguments.

You can represent the equality constraint $x_1 + 2x_2 + 4x_3 = 7$ in matrix form as

$$Aeq \cdot x = beq$$

where *Aeq* = [1 2 4] and *beq* = [7]. To create variables for *Aeq* and *beq*, enter

```
Aeq = [1 2 4];
beq = [7];
```

Finding the Solution

To solve the optimization problem, enter

```
[x, fval] =lsqlin(C, d, [], [], Aeq, beq)
```

`lsqlin` returns

```
x =

    0.3333
    0.6667
    1.3333

fval =

    2.3333
```

The minimum occurs at the point `x` and `fval` is the square of the distance from `x` to the origin.

Note In this example, `lsqlin` issues a warning that it is switching from its default *large-scale* algorithm to its *medium-scale* algorithm. This message has no bearing on the result, so you can safely ignore it. "Using the Optimization Functions" on page 2-6 provides more information on large and medium-scale algorithms.

More Examples

The following sections contain more examples of solving optimization problems:

- "Examples That Use Standard Algorithms" on page 2-9
- "Large-Scale Examples" on page 2-40

2

Tutorial

The Tutorial provides information on how to use the toolbox functions. It also provides examples for solving different optimization problems. It consists of these sections.

Introduction (p. 2-3)	Summarizes, in tabular form, the functions available for minimization, equation solving, and solving least-squares or data fitting problems. It also provides basic guidelines for using the optimization routines and introduces the algorithms and line-search strategies that are available for solving medium- and large-scale problems.
Examples That Use Standard Algorithms (p. 2-9)	Presents *medium-scale* algorithms through a selection of minimization examples. These examples include unconstrained and constrained problems, as well as problems with and without user-supplied gradients. This section also discusses maximization, greater-than-zero constraints, passing additional arguments, and multiobjective examples.
Large-Scale Examples (p. 2-40)	Presents *large-scale* algorithms through a selection of large-scale examples. These examples include specifying sparsity structures, and preconditioners, as well as unconstrained and constrained problems.
Default Options Settings (p. 2-76)	Describes the use of default options settings and tells you how to change them. It also tells you how to determine which options are used by a specified function, and provides examples of setting some commonly used options.
Displaying Iterative Output (p. 2-79)	Describes the column headings used in the iterative output of both medium-scale and large-scale algorithms.
Calling an Output Function Iteratively (p. 2-85)	Describes how to make an optimization function call an output function at each iteration.

Optimizing Anonymous Functions Instead of M-Files (p. 2-90)	Tells you how to represent a mathematical function at the command line by creating an anonymous function from a string expression.
Typical Problems and How to Deal with Them (p. 2-92)	Provides tips to help you improve solutions found using the optimization functions, improve efficiency of the algorithms, overcome common difficulties, and transform problems that are typically not in standard form.
Selected Bibliography (p. 2-96)	Lists published materials that support concepts implemented in the Optimization Toolbox.

Introduction

Optimization is the process of finding the minimum or maximum of a function, usually called the *objective function*. 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.

This introduction includes the following sections:

- Problems Covered by the Toolbox
- Using the Optimization Functions

Problems Covered by the Toolbox

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

Note The following tables list the types of problems in order of increasing complexity.

Table 2-1: Minimization

Type	Notation	Function
Scalar Minimization	$\min_{a} f(a)$ such that $a_1 \le a \le a_2$	`fminbnd`
Unconstrained Minimization	$\min_{x} f(x)$	`fminunc`, `fminsearch`
Linear Programming	$\min_{x} f^T x$ such that $A \cdot x \le b,\ \ Aeq \cdot x = beq,\ \ l \le x \le u$	`linprog`

Table 2-1: Minimization (Continued)

Type	Notation	Function
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 $c(x) \le 0, \quad ceq(x) = 0$ $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u$	`fmincon`
Goal Attainment	$\min_{x, \gamma} \gamma$ such that $F(x) - w\gamma \le \text{goal}$ $c(x) \le 0, \quad ceq(x) = 0$ $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u$	`fgoalattain`
Minimax	$\min_{x} \max_{\{F_i\}} \{F_i(x)\}$ such that $c(x) \le 0, \quad ceq(x) = 0$ $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u$	`fminimax`
Semi-Infinite Minimization	$\min_{x} f(x)$ such that $K(x, w) \le 0$ for all w $c(x) \le 0, \quad ceq(x) = 0$ $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u$	`fseminf`

Table 2-2: 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 2-3: Least-Squares (Curve Fitting)

Type	Notation	Function
Linear Least-Squares	$\min\limits_{x} \lVert C \cdot x - d \rVert_2^2$, m equations, n variables	`\ (slash)`
Nonnegative Linear-Least-Squares	$\min\limits_{x} \lVert C \cdot x - d \rVert_2^2$ such that $x \geq 0$	`lsqnonneg`
Constrained Linear-Least-Squares	$\min\limits_{x} \lVert C \cdot x - d \rVert_2^2$ such that $A \cdot x \leq b, \quad Aeq \cdot x = beq, \quad l \leq x \leq u$	`lsqlin`
Nonlinear Least-Squares	$\min\limits_{x} \frac{1}{2} \lVert F(x) \rVert_2^2 = \frac{1}{2} \sum_i F_i(x)^2$ such that $l \leq x \leq u$	`lsqnonlin`
Nonlinear Curve Fitting	$\min\limits_{x} \frac{1}{2} \lVert F(x, xdata) - ydata \rVert_2^2$ such that $l \leq x \leq u$	`lsqcurvefit`

Using the Optimization Functions

This section provides some basic information about using the optimization functions.

Defining the Objective Function

Many of the optimization functions require you to create a MATLAB function that computes the objective function. The function should accept vector inputs and return a scalar output of type `double`. There are two ways to create the objective function:

- Create an anonymous function at the command line. For example, to create an anonymous function for x^2, enter

  ```
  square = @(x) x.^2;
  ```

  You then call the optimization function with `square` as the first input argument. You can use this method if the objective function is relatively simple and you do not need to use it again in a future MATLAB session.

- Write an M-file for the function. For example, to write the function x^2 as a M-file, open a new file in the MATLAB editor and enter the following code:

  ```
  function y = square(x)
  y = x.^2;
  ```

  You can then call the optimization function with `@square` as the first input argument. The `@` sign creates a function handle for `square`. Use this method if the objective function is complicated or you plan to use it for more than one MATLAB session.

Note The functions in the Optimization Toolbox only accept inputs of type `double`, so the objective function must return a scalar output of type `double`.

Maximizing Versus Minimizing

The optimization functions in the toolbox minimize the objective function. To maximize a function *f*, apply an optimization function to minimize -*f*. The resulting point where the maximum of *f* occurs is also the point where the minimum of -*f* occurs.

Changing Options

You can change the default options for an optimization function by passing in an `options` structure, which you create using the function `optimset`, as an input argument. See “Default Options Settings” on page 2-76 for more information.

Supplying the Gradient

Many of the optimization functions use the gradient of the objective function to search for the minimum. You can write a function that computes the gradient and pass it to an optimization function using the `options` structure. “Constrained Example with Gradients” on page 2-14 provides an example of how to do this. Providing a gradient function improves the accuracy and speed of the optimization function. However, for some objective functions it might not be possible to provide a gradient function, in which case the optimization function calculates it using an adaptive finite-difference method.

Medium- and Large-Scale Algorithms

This guide separates “medium-scale” algorithms from “large-scale” algorithms. Medium-scale is not a standard term and is used here only to distinguish 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 (Broyden, Fletcher, Goldfarb, and Shanno) quasi-Newton method. For constrained minimization, minimax, goal attainment, and semi-infinite optimization, variations of *sequential quadratic programming* (SQP) are used. Nonlinear least-squares problems use the Gauss-Newton and Levenberg-Marquardt methods. Nonlinear equation solving also uses the trust-region dogleg algorithm.

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.

Examples That Use Standard Algorithms

This section presents the *medium-scale* (i.e., standard) algorithms through a tutorial. Examples similar to those in the first part of this tutorial ("Unconstrained Minimization Example" through the "Equality Constrained Example") can also be found in the first demonstration, "Tutorial Walk Through," in the M-file `optdemo`.

Note Medium-scale is not a standard term and is used to differentiate these algorithms from the large-scale algorithms described in "Large-Scale Algorithms" on page 4-1.

The tutorial uses the functions `fminunc`, `fmincon`, and `fsolve`. The other optimization routines, `fgoalattain`, `fminimax`, `lsqnonlin`, and `fseminf`, are used in a nearly identical manner, with differences only in the problem formulation and the termination criteria. The section "Multiobjective Examples" on page 2-27 discusses multiobjective optimization and gives several examples using `lsqnonlin`, `fminimax`, and `fgoalattain`, including how Simulink can be used in conjunction with the toolbox.

This section includes the following examples:

- Unconstrained Minimization Example
- Nonlinear Inequality Constrained Example
- Constrained Example with Bounds
- Constrained Example with Gradients
- Gradient Check: Analytic Versus Numeric
- Equality Constrained Example

It also discusses

- Maximization
- Greater-Than-Zero Constraints
- Avoiding Global Variables via Anonymous and Nested Functions
- Nonlinear Equations with Analytic Jacobian

- Nonlinear Equations with Finite-Difference Jacobian
- Multiobjective Examples

Unconstrained Minimization Example

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

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

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 whether 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 more than one local minimum exists, 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 preceding example, `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, whether a user-supplied gradient or Jacobian is to be used, and the maximum number of iterations or function evaluations. See `optimset`, the individual optimization functions, and "Optimization Options" on page 5-9 for more options and information.

Nonlinear Inequality Constrained Example

If inequality constraints are added to Eq. 2-1, the resulting problem can be solved by the `fmincon` function. For example, find x that solves

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

subject to the constraints

$$x_1x_2 - x_1 - x_2 \le -1.5$$
$$x_1x_2 \ge -10$$

Because 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

$$\begin{aligned} x_1x_2 - x_1 - x_2 + 1.5 &\le 0 \\ -x_1x_2 - 10 &\le 0 \end{aligned} \qquad (2\text{-}3)$$

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
```

You can evaluate the constraints at the solution by entering

```
[c,ceq] = confun(x)
```

This returns

```
c=
    1.0e-14 *
    0.1110
   -0.1776

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. 2-2 to be greater than zero (i.e., $x_1 \ge 0$, $x_2 \ge 0$), 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 there are 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 $x_1 \le 10$, $0 \le x_2$ (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 because 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 Eq. 2-2 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`:

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

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

$$\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_2}{\partial x_1} \\ \frac{\partial c_1}{\partial x_2} & \frac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix} \tag{2-5}$$

Since you are providing the gradient of the objective in `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 options `GradObj` and `GradConstr` to `'on'` in the example's existing `options` structure:

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

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

The arguments `lb` and `ub` place lower and upper bounds on the independent variables in `x`. In this example, there are 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 you want such gradient checks, set the `DerivativeCheck` option 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 solve the problem using the following steps.

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 that nonlinear inequality constraints are of the form $C_i(x) \le 0$. Greater-than-zero constraints are expressed as less-than-zero constraints by multiplying them by -1. For example, a constraint of the form $C_i(x) \ge 0$ is equivalent to the constraint $(-C_i(x)) \le 0$; a constraint of the form $C_i(x) \ge b$ is equivalent to the constraint $(-C_i(x) + b) \le 0$.

Parameterizing Your Function as a Nested Functions

As an alternative to writing your function as an anonymous function, you can write a single M-file that

- Accepts the additional parameters to your function as inputs.
- Invokes the optimization function.
- Contains your function as a nested function.

The following example illustrates how to write an M-file to find zeros of the $x^3 + bx + c$, for different values of the coefficients b and c.

```
function y = findzero(b, c, x0)

options = optimset('Display', 'off'); % Turn off Display
y = fsolve(@poly, x0, options);

   function y = poly(x) % Compute the polynomial.
   y = x^3 + b*x + c;
   end
end
```

The main function, `findzero`, does two things:

- Invokes the function `fzero` to find a zero of the polynomial.
- Computes the polynomial in a nested function, `poly`, which is called by `fzero`.

You can call `findzero` with any values of the coefficients `b` and `c`, which are then automatically passed to `poly` because it is a nested function.

As an example, to find a zero of the polynomial with `b = 2` and `c = 3.5`, using the starting point `x0 = 0`, call `findzero` as follows.

```
x = findzero(2, 3.5, 0)
```

This returns the zero

```
x =

   -1.0945
```

Avoiding Global Variables via Anonymous and Nested Functions

The optimization functions in the toolbox use several types of functions that you define, including

- The objective function
- The constraint functions (for `fmincon`, `fseminf`, `fgoalattain`, and `fminimax`)

- The Hessian and Jacobian multiply functions, `HessMult` and `JacobMult` respectively, for the large-scale `fmincon`, `fminunc`, `lsqnonlin`, `lsqcurvefit` and `fsolve`
- An output function

Sometimes these functions might require some additional parameters besides the independent variable. There are two ways to provide these additional parameters to the function:

- Parameterize your function and then create a function handle to an anonymous function that calls your function. This is explained in "Parameterizing Your Function Using an Anonymous Function" on page 2-20
- Write your function as a nested function within an outer function that calls the solver. This method has the additional advantage that you can share variables between your functions, as explained in "Parameterizing Your Function as a Nested Functions" on page 2-18.

Parameterizing Your Function Using an Anonymous Function

As an example, suppose you want to find the zeros of the function `ellipj` using `fsolve`. `fsolve` expects the objective function to take one input argument, but the `ellipj` function takes two, `u` and `m`. You can see this function by typing

```
type ellipj
```

You are solving for the variable `u`, whiled `m` is simply a second parameter to specify which Jacobi elliptic function. To look for a zero near `u0 = 3` for `m = 0.5`, you can create a function handle to an anonymous function that captures the current value of `m` from the workspace. Then, when the solver `fsolve` calls this function handle, the parameter `m` exists and `ellipj` will be called with two arguments. You pass this function handle to `fsolve` with the following commands.

```
u0 = 3;
m = 0.5;
options = optimset('Display','off'); % Turn off Display
x = fsolve(@(u) ellipj(u,m), u0, options)

x =
      3.7081
```

Sharing Variables Using Nested Functions

The preceding example uses an existing function `ellipj` that has more arguments than would be passed by `fsolve`. If you are writing your own function, you can use the technique above, or you might find it more convenient to use a nested function. Nested functions have the additional advantage that you can share variables between them. For example, suppose you want to minimize an objective function, subject to an additional nonlinear constraint that the objective function should never exceed a certain value. To avoid having to recompute the objective function value in the constraint function, you can use a nested function. The following code illustrates this.

```
function [x,fval] = runsharedvalues(a,b,c,d,lower)

objval = []; % Initialize shared variable
x0 = [-1,1];     % Make a starting guess at the solution
options = optimset('LargeScale','off');
[x, fval] =
fmincon(@objfun,x0,[],[],[],[],[],[],@constrfun,options);

  function f = objfun(x)
    % Nonlinear objective function
    % Variable objval shared with objfun and runsharedvalues
    objval = exp(x(1))*(a*x(1)^2+b*x(2)^2+c*x(1)*x(2)+d*x(2)+1);
    f = objval;
  end

  function [c,ceq] = constrfun(x)
    % Nonlinear inequality constraints
    % Variable objval shared with objfun and runsharedvalues
    c(1) = - objval + lower;
    c(2:3) = [1.5 + x(1)*x(2) - x(1) - x(2); -x(1)*x(2) - 10];
    % Nonlinear equality constraints
    ceq = [];

  end
end
```

Now you can run the objective with different values for `a`, `b`, `c`, and `d` and the first nonlinear constraint will ensure you always find the minimum of the objective where the objective never goes below the value of `lower`.

```
[x, fval ] = runsharedvalues(-4, 2, 4, 2, 1)

Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                     1

x =

   -0.6507    1.3030


fval =

     1




[x,fval] = runsharedvalues(4, 2, 4, 2, 0.5)
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                     1

x =

   -4.9790    1.9664


fval =

    0.5000
```

You can see another example of sharing variables via nested functions in "Simulink Example Using fminimax" on page 2-33.

Nonlinear Equations with Analytic Jacobian

This example demonstrates the use of the default medium-scale `fsolve` algorithm. It is intended for problems where

- The system of nonlinear equations is square, i.e., the number of equations equals the number of unknowns.
- There exists a solution x such that $F(x) = 0$.

The example uses `fsolve` to obtain the minimum of the banana (or Rosenbrock) function by deriving and then solving an equivalent system of nonlinear equations. The Rosenbrock function, which has a minimum at $F(x) = 0$, is a common test problem in optimization. It has a high degree of nonlinearity and converges extremely slowly if you try to use steepest descent type methods. It is given by

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

First generalize this function to an n-dimensional function, for any positive, even value of n:

$$f(x) = \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2$$

This function is referred to as the generalized Rosenbrock function. It consists of n squared terms involving n unknowns.

Before you can use `fsolve` to find the values of x such that $F(x) = 0$, i.e., obtain the minimum of the generalized Rosenbrock function, you must rewrite the function as the following equivalent system of nonlinear equations:

$$
\begin{aligned}
F(1) &= 1 - x_1 \\
F(2) &= 10(x_2 - x_1^2) \\
F(3) &= 1 - x_3 \\
F(4) &= 10(x_4 - x_3^2) \\
&\vdots \\
F(n-1) &= 1 - x_{n-1} \\
F(n) &= 10(x_n - x_{n-1}^2)
\end{aligned}
$$

This system is square, and you can use `fsolve` to solve it. As the example demonstrates, this system has a unique solution given by $x_i = 1, i = 1, \ldots, n$.

Step 1: Write an M-file bananaobj.m to compute the objective function values and the Jacobian.

```
function [F,J] = bananaobj(x);
% Evaluate the vector function and the Jacobian matrix for
% the system of nonlinear equations derived from the general
% n-dimensional Rosenbrock function.
% Get the problem size
n = length(x);
if n == 0, error('Input vector, x, is empty.'); end
if mod(n,2) ~= 0,
   error('Input vector, x, must have an even number of
components.');
end
% Evaluate the vector function
odds  = 1:2:n;
evens = 2:2:n;
F = zeros(n,1);
F(odds,1)  = 1-x(odds);
F(evens,1) = 10.*(x(evens)-x(odds).^2);
% Evaluate the Jacobian matrix if nargout > 1
if nargout > 1
   c = -ones(n/2,1);    C = sparse(odds,odds,c,n,n);
   d = 10*ones(n/2,1);  D = sparse(evens,evens,d,n,n);
```

```
   e = -20.*x(odds);    E = sparse(evens,odds,e,n,n);
   J = C + D + E;
end
```

Step 2: Call the solve routine for the system of equations.

```
n = 64;
x0(1:n,1) = -1.9;
x0(2:2:n,1) = 2;
options=optimset('Display','iter','Jacobian','on');
[x,F,exitflag,output,JAC] = fsolve(@bananaobj,x0,options);
```

Use the starting point $x(i) = -1.9$ for the odd indices, and $x(i) = 2$ for the even indices. Accept the `fsolve` default `'off'` for the `LargeScale` option, and the default medium-scale nonlinear equation algorithm `'dogleg'`. Then set `Jacobian` to `'on'` to use the Jacobian defined in `bananaobj.m` . The `fsolve` function generates the following output:

```
                                         Norm of  First-order Trust-region
Iteration Func-count   f(x)                 step   optimality       radius
    0         1       4281.92                             615            1
    1         2       1546.86               1             329            1
    2         3       112.552             2.5            34.8          2.5
    3         4        106.24            6.25            34.1         6.25
    4         5        106.24            6.25            34.1         6.25
    5         6       51.3854          1.5625            6.39         1.56
    6         7       51.3854         3.90625            6.39         3.91
    7         8       43.8722        0.976562            2.19        0.977
    8         9       37.0713         2.44141            6.27         2.44
    9        10       37.0713         2.44141            6.27         2.44
   10        11       26.2485        0.610352            1.52         0.61
   11        12       20.6649         1.52588            4.63         1.53
   12        13       17.2558         1.52588            6.97         1.53
   13        14       8.48582         1.52588            4.69         1.53
   14        15       4.08398         1.52588            3.77         1.53
   15        16       1.77589         1.52588            3.56         1.53
   16        17      0.692381         1.52588            3.31         1.53
   17        18      0.109777         1.16206            1.66         1.53
   18        19             0       0.0468565               0         1.53
Optimization terminated successfully:
 First-order optimality is less than options.TolFun
```

Nonlinear Equations with Finite-Difference Jacobian

In the preceding example, the function `bananaobj` evaluates `F` and computes the Jacobian `J`. What if the code to compute the Jacobian is not available? By default, if you do not indicate that the Jacobian can be computed in the objective function (by setting the `Jacobian` option in `options` to `'on'`), `fsolve`, `lsqnonlin`, and `lsqcurvefit` instead use finite differencing to approximate the Jacobian. This is the default Jacobian option.You can select finite differencing by setting `Jacobian` to `'off'` using `optimset`.

This example uses `bananaobj` from the preceding example as the objective function, but sets `Jacobian` to `'off'` so that `fsolve` approximates the Jacobian and ignores the second `bananaobj` output. It accepts the `fsolve` default `'off'` for the `LargeScale` option, and the default nonlinear equation medium-scale algorithm `'dogleg'`:

```
n = 64;
x0(1:n,1) = -1.9;
x0(2:2:n,1) = 2;
options=optimset('Display','iter','Jacobian','off');
[x,F,exitflag,output,JAC] = fsolve(@bananaobj,x0,options);
```

The example produces the following output:

```
                                    Norm of  First-order Trust-region
Iteration Func-count   f(x)            step   optimality       radius
     0         65       4281.92                      615             1
     1        130       1546.86             1        329             1
     2        195       112.552           2.5       34.8           2.5
     3        260        106.24          6.25       34.1          6.25
     4        261        106.24          6.25       34.1          6.25
     5        326       51.3854        1.5625       6.39          1.56
     6        327       51.3854       3.90625       6.39          3.91
     7        392       43.8722      0.976562       2.19         0.977
     8        457       37.0713       2.44141       6.27          2.44
     9        458       37.0713       2.44141       6.27          2.44
    10        523       26.2485      0.610352       1.52          0.61
    11        588       20.6649       1.52588       4.63          1.53
    12        653       17.2558       1.52588       6.97          1.53
    13        718       8.48582       1.52588       4.69          1.53
    14        783       4.08398       1.52588       3.77          1.53
    15        848       1.77589       1.52588       3.56          1.53
```

```
    16      913    0.692381     1.52588       3.31          1.53
    17      978    0.109777     1.16206       1.66          1.53
    18     1043           0   0.0468565          0          1.53
 Optimization terminated successfully:
  First-order optimality is less than options.TolFun
```

The finite-difference version of this example requires the same number of iterations to converge as the analytic Jacobian version in the preceding example. It is generally the case that both versions converge at about the same rate in terms of iterations. However, the finite-difference version requires many additional function evaluations. The cost of these extra evaluations might or might not be significant, depending on the particular problem.

Multiobjective Examples

The previous examples involved problems with a single objective function. This section shows how to solve problems with multiobjective functions using `lsqnonlin`, `fminimax`, and `fgoalattain`. The first two examples show how to optimize parameters in a Simulink model.

This section presents the following examples:

- "Simulink Example Using lsqnonlin" on page 2-27
- "Simulink Example Using fminimax" on page 2-33
- "Signal Processing Example" on page 2-36

Simulink Example Using lsqnonlin

Suppose that you want to optimize the control parameters in the Simulink model `optsim.mdl`. (This model can be found in the Optimization Toolbox `optim` 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 2-1, Plant with Actuator Saturation.

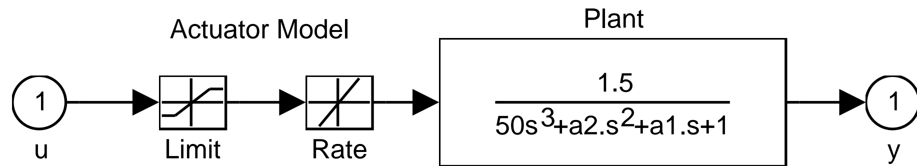


Figure 2-1: Plant with Actuator Saturation

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 closed-loop response of the system to a step input is shown in Figure 2-2, Closed-Loop Response. You can see this response by opening the model (type `optsim` at the command line or click the model name), and selecting **Start** from the **Simulation** menu. The response plots to the scope.

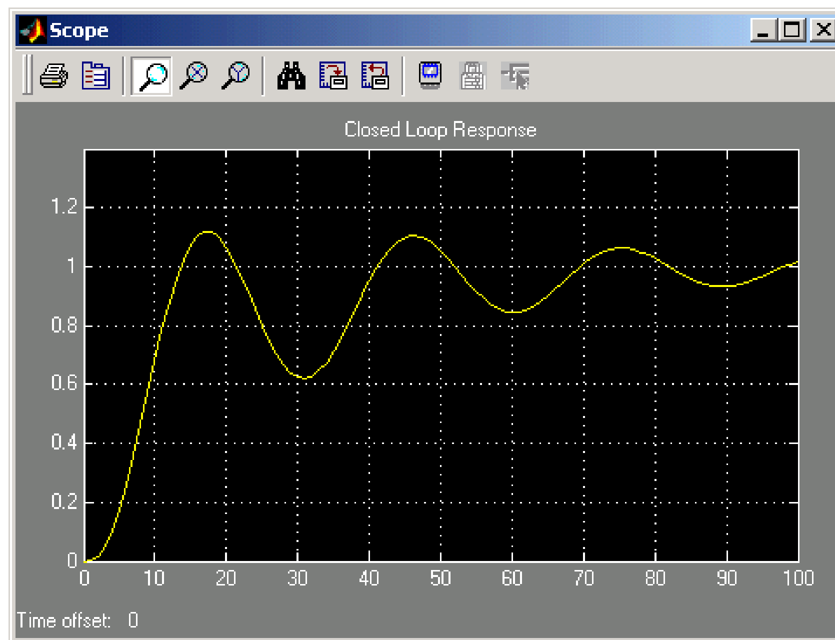


Figure 2-2: Closed-Loop Response

The problem is to design a feedback control loop 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 2-3, Closed-Loop Model.

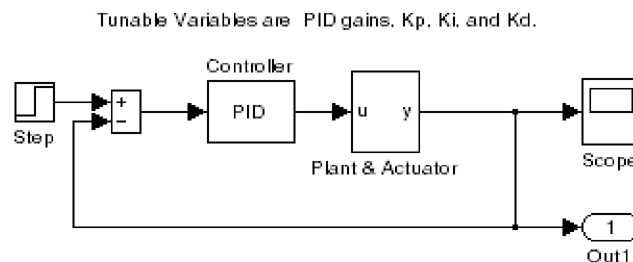


Figure 2-3: Closed-Loop Model

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 Proportional Integral Derivative (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. The tracking is performed via an M-file function `tracklsq`, which returns the error signal `yout`, the output computed by calling `sim`, minus the input signal `1`. The code for `tracklsq`, shown below, is contained in the file `runtracklsq.m`, which is included in the Optimization Toolbox.

The function `runtracklsq` sets up all the needed values and then calls `lsqnonlin` with the objective function `tracklsq`, which is nested inside `runtracklsq`. 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`.

To run the simulation in the model `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 to be optimized. The function `tracklsq` is nested inside `runtracklsq` so that the variables `a1` and `a2` are shared between the two functions. The variables `a1` and `a2` are initialized in `runtracklsq`.

The objective function `tracklsq` must run the simulation. The simulation can be run either in the base workspace or the current workspace, that is, the workspace of the function calling `sim`, which in this case is the workspace of `tracklsq`. In this example, the `simset` command is used to tell `sim` to run the simulation in the current workspace by setting `'SrcWorkspace'` to `'Current'`. You can also choose a solver for `sim` using the `simset` function. The simulation is performed using a fixed-step fifth-order method to `100` seconds.

When the simulation is completed, the variables `tout`, `xout`, and `yout` are now in the current workspace (that is, the workspace of `tracklsq`). The Outport block in the block diagram model puts `yout` into the current workspace at the end of the simulation.

The following is the code for `runtracklsq`.

```
function [Kp,Ki,Kd] = runtracklsq
% RUNTRACKLSQ demonstrates using LSQNONLIN with Simulink.
```

```
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);
Kp = pid(1); Ki = pid(2); Kd = pid(3);

    function F = tracklsq(pid)
      % Track the output of optsim to a signal of 1

      % Variables a1 and a2 are needed by the model optsim.
      % They are shared with RUNTRACKLSQ so do not need to be
      % redefined here.
      Kp = pid(1);
      Ki = pid(2);
      Kd = pid(3);

      % Compute function value
      simopt = simset('solver','ode5','SrcWorkspace','Current');
      % Initialize sim options
      [tout,xout,yout] = sim('optsim',[0 100],simopt);
      F = yout-1;

    end
end
```

When you run `runtracklsq`, the optimization gives the solution for the proportional, integral, and derivative (`Kp`, `Ki`, `Kd`) gains of the controller after 64 function evaluations.

```
[Kp, Ki, Kd] = runtracklsq

                                                       Directional
 Iteration  Func-count    Residual     Step-size       derivative    Lambda
     0           4        8.66531
     1          18        5.21604          85.4         -0.00836       6.92469
     2          25        4.53699             1           -0.107     0.0403059
     3          32        4.47316         0.973         -0.00209     0.0134348
     4          40        4.46854          2.45        9.72e-005    0.00676229
     5          47        4.46575         0.415         -0.00266    0.00338115
     6          48        4.46526             1        -0.000999    0.00184785
Optimization terminated: directional derivative along
```

```
 search direction less than TolFun and infinity-norm of
 gradient less than 10*(TolFun+TolX).

Kp =

    3.0956


Ki =

    0.1466


Kd =

   14.1378
```

The resulting closed-loop step response is shown in Figure 2-4.

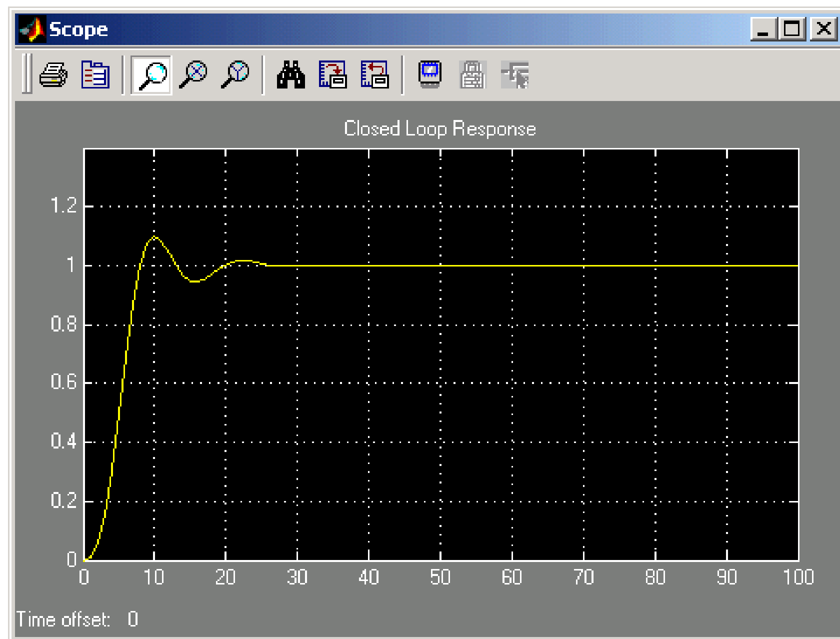


Figure 2-4: Closed-Loop Response Using lsqnonlin

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 might 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 can 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 [1].)

The Nonlinear Control Design 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 a problem of lack of smoothness.

Simulink Example Using fminimax

Another approach to optimizing the control parameters in the Simulink model shown in “Plant with Actuator Saturation” on page 2-28 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.

The code for this example, shown below, is contained in the function `runtrackmm`, in which the objective function is simply the output `yout` returned by the `sim` command. But minimizing the maximum output at all time steps might force the output to be far below unity for some time steps. To keep the output above `0.95` after the first `20` seconds, the constraint function `trackmmcon` contains the 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. This way you can avoid calling the

simulation twice by using nested functions so that the value of yout can be shared between the objective and constraint functions as long as it is initialized in runtrackmm.

The following is the code for runtrackmm.

```
function [Kp, Ki, Kd] = runtrackmm
%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.4.6.1 $  $Date: 2004/02/11 14:43:27 $

optsim
pid0 = [0.63 0.0504 1.9688];
% a1, a2, yout are shared with TRACKMMOBJ and TRACKMMCON
a1 = 3; a2 = 43; % Initialize plant variables in model
yout = []; % Give yout an initial value
options = optimset('Display','iter',...
    'TolX',0.001,'TolFun',0.001);
pid = fminimax(@trackmmobj,pid0,[],[],[],[],[],[],...
    @trackmmcon,options);
Kp = pid(1); Ki = pid(2); Kd = pid(3);

    function F = trackmmobj(pid)
        % Track the output of optsim to a signal of 1
        % Variables a1 and a2 are shared with RUNTRACKMM.
        % Variable yout is shared with RUNTRACKMM and
        % RUNTRACKMMCON.

        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;
    end

    function [c,ceq] = trackmmcon(pid)
        % Track the output of optsim to a signal of 1
        % Variable yout is shared with RUNTRACKMM and
        % TRACKMMOBJ
```

```
            % Compute constraints.
            % We know objective function TRACKMMOBJ is called before
            % this
            %  constraint function and so yout is current.
            c = -yout(20:100)+.95;
            ceq=[];
        end
    end
```

When you run the code, it returns the following results.

```
[Kp,Ki,Kd] = runtrackmm

                        Max                     Directional
 Iter   F-count  {F,constraints}  Step-size     derivative    Procedure
    0      5        1.11982
    1     11          1.264            1            1.18
    2     17          1.055            1          -0.172
    3     23          1.004            1         -0.0128    Hessian modified twice
    4     29         0.9997            1       3.48e-005    Hessian modified
    5     35         0.9996            1      -1.36e-006    Hessian modified twice
Optimization terminated: Search direction less than 2*options.TolX
 and maximum constraint violation is less than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                      1
                                     14
                                    182

Kp =

    0.5894


Ki =

    0.0605


Kd =

    5.5295
```

The last value shown in the `MAX{F,constraints}` column of the output shows that the maximum value for all the time steps is `0.9996`. The closed loop response with this result is shown in the following Figure 2-5, Closed-Loop Response Using fminimax.

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

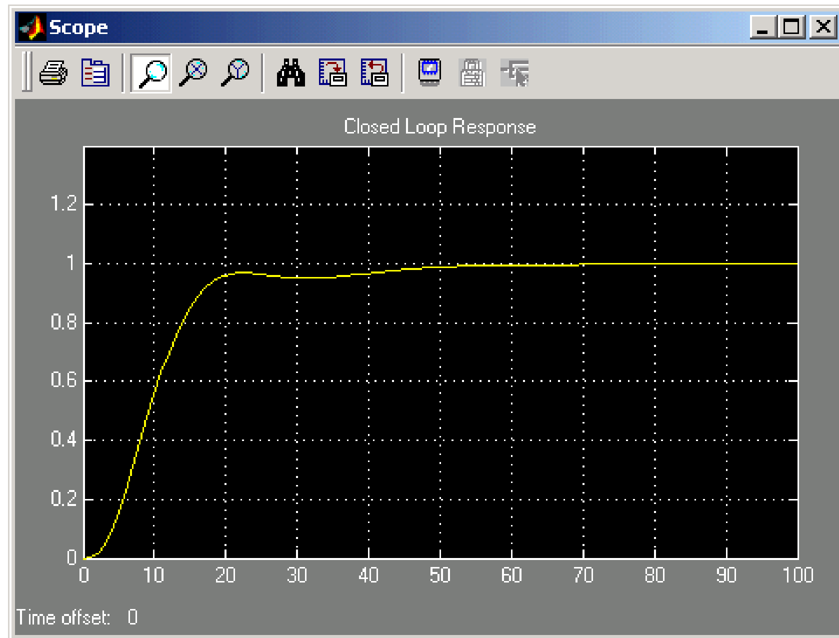


Figure 2-5: Closed-Loop Response Using fminimax

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

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

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 “Goal Attainment Method” on page 3-47). 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,'.r');
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(@(x) filtmin(x,w0)...
    a0,goal,weight,[],[],[],[],[],[],[],options);

% 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 2-6). Note that you could use the `remez` function in the Signal Processing Toolbox to design this filter.

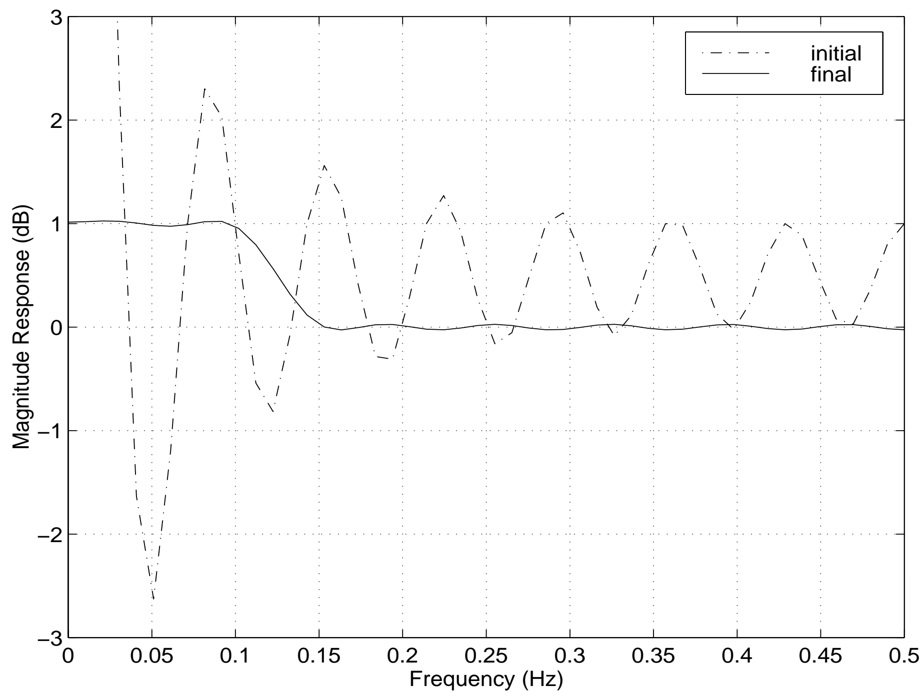


Figure 2-6: Magnitude Response with Initial and Final Magnitude Coefficients

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 that the gradient be computed in the user-supplied function).

This section summarizes the kinds of problems covered by large-scale methods and provides these examples:

- Nonlinear Equations with Jacobian
- Nonlinear Equations with Jacobian Sparsity Pattern
- Nonlinear Least-Squares with Full Jacobian Sparsity Pattern
- Nonlinear Minimization with Gradient and Hessian
- Nonlinear Minimization with Gradient and Hessian Sparsity Pattern
- Nonlinear Minimization with Bound Constraints and Banded Preconditioner
- Nonlinear Minimization with Equality Constraints
- Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints
- Quadratic Minimization with Bound Constraints
- Quadratic Minimization with a Dense but Structured Hessian
- Linear Least-Squares with Bound Constraints
- Linear Programming with Equalities and Inequalities
- Linear Programming with Dense Columns in the Equalities

Problems Covered by Large-Scale Methods

This section describes how to formulate problems for functions that use large-scale methods. It is important to keep in mind that there are some restrictions on the types of problems covered by large-scale methods. For example, the function `fmincon` cannot use large-scale methods when the feasible region is defined by either of the following:

- Nonlinear equality or inequality constraints
- Both upper- or lower-bound constraints and equality constraints

When a function is unable to solve a problem using large-scale methods, it reverts to medium-scale methods.

Formulating Problems with Large-Scale Methods

The following table summarizes how to set up problems for large-scale methods and provide the necessary input for the optimization functions. For each function, the second column of the table describes how to formulate the problem and 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.

Note The following table lists the functions in order of increasing problem complexity.

Several examples, which follow this table, clarify the contents of the table.

Table 2-4: 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 $Aeq \cdot x = beq$, and 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. • Aeq 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).	None	• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.

Table 2-4: Large-Scale Problem Coverage and Requirements (Continued)

Function	Problem Formulations	Additional Information Needed	For Large Problems
`lsqcurvefit`	• $\min_x \frac{1}{2}\\|F(x, xdata) - ydata\\|_2^2$ • $\min_x \frac{1}{2}\\|F(x, xdata) - ydata\\|_2^2$ such that $l \le x \le u$ where $l < u$ $F(x, xdata)$ must be overdetermined (have at least as many equations as variables).	None	• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.
`fsolve`	$F(x) = 0$ $F(x)$ must have the same number of equations as variables.	None	• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.
`lsqlin`	$\min_x \\|C \cdot x - d\\|_2^2$ 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.	None	C should be sparse.
`linprog`	$\min_x f^T x$ such that $A \cdot x \le b$ and $Aeq \cdot x = beq$, where $l \le x \le u$	None	A and Aeq should be sparse.

Table 2-4: Large-Scale Problem Coverage and Requirements (Continued)

Function	Problem Formulations	Additional Information Needed	For Large Problems
`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 $Aeq \cdot x = beq$, and Aeq is an m-by-n matrix where $m \le n$.	None	• H should be sparse. • Aeq should be sparse.

In the following examples, 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` determines 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 must 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 solve routine for the system of equations.

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

A starting point is given as well as the function name. The default method for `fsolve` is medium-scale, so it is necessary to specify `'LargeScale'` as `'on'` in the `options` argument. Setting the `Display` option to `'iter'` causes `fsolve` to display the output at each iteration. Setting `Jacobian` to `'on'`, causes `fsolve` to use the Jacobian information available in `nlsf1.m`.

The commands display this output:

```
                                           Norm of  First-order   CG-
Iteration  Func-count     f(x)              step    optimality  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 `PrecondBandWidth` is 0 in `options`, so a diagonal preconditioner is used. (`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 conjugate gradient (CG) iterations required per major iteration is low, at most five for a problem of 1000 dimensions, implying that 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 option `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',...
                   'LargeScale','on','PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case the output is

```
                                    Norm of  First-order  CG-
Iteration Func-count     f(x)          step  optimality  Iterations
     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. See “Preconditioned Conjugate Gradients” on page 4-5.

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 that the Jacobian can be computed in `nlsf1` (by setting the `Jacobian` option in `options` to `'on'`), `fsolve`, `lsqnonlin`, and `lsqcurvefit` instead uses finite differencing to approximate the Jacobian.

In order for this finite differencing to be as efficient as possible, you should supply the sparsity pattern of the Jacobian, by setting `JacobPattern` to `'on'` in `options`. That is, supply a sparse matrix `Jstr` whose nonzero entries correspond to nonzeros of the Jacobian for all x. Indeed, the nonzeros 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 nonzeros 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 the objective function either) then, in this problem `nlsfs1`, the finite-differencing code attempts 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 attempts to compute. In other words, this problem is solvable if you provide the sparsity pattern. If not, most computers 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 that 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,...
                   'LargeScale','on','PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case, the output displayed is

```
                                          Norm of First-order  CG-
Iteration Func-count    f(x)          step  optimality  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     0.0631898      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 you set `PrecondBandWidth` to `Inf`, then a sparse direct linear solver is used instead of a preconditioned conjugate gradient iteration:

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

and the resulting display is

```
                                          Norm of First-order  CG-
Iteration Func-count    f(x)          step  optimality  Iterations
    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     1.73538e-008     0.0397925   0.000196        1
     5       26     1.13169e-018 4.55544e-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 is 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 might 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 finding 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
```

Because the Jacobian is not computed in `myfun.m` , and no Jacobian sparsity pattern is provided by the `JacobPattern` option in `options`, `lsqnonlin` calls the large-scale method with `JacobPattern` set to `Jstr = sparse(ones(10,2))`. This is the default for `lsqnonlin`. Note that the `Jacobian` option in `options` is set to `'off'` by default.

When the finite-differencing routine is called the first time, it detects that `Jstr` is actually a dense matrix, i.e., that no speed benefit is derived from storing it as a sparse matrix. From then on the finite-differencing routine uses `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 can 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 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

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

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 long 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 that this information is available in `brownfgh`, using the `GradObj` and `Hessian` options.

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 infinity norm of 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, 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)
% 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` option to `Hstr`. When a problem as large as this has obvious sparsity structure, not setting the `HessPattern` option requires a huge amount of unnecessary memory and computation because `fminunc` attempts to use finite differencing on a full Hessian matrix of one million nonzero entries.

You must also set the `GradObj` option to `'on'` using `optimset`, since the gradient is computed in `brownfg.m`. Then execute `fminunc` as shown in Step 2.

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 eight iterations and seven 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 infinity norm of 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 $x_0 = x_{n+1} = 0$.

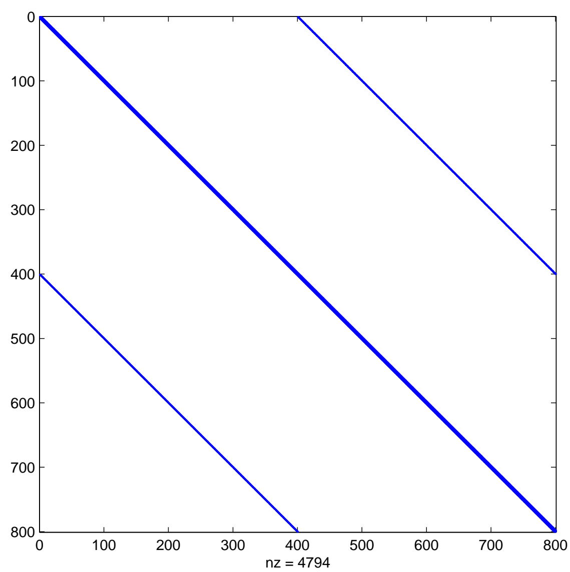
Step 1: Write an M-file tbroyfg.m that computes the objective function and the gradient of the objective

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

```
type tbroyfg
```

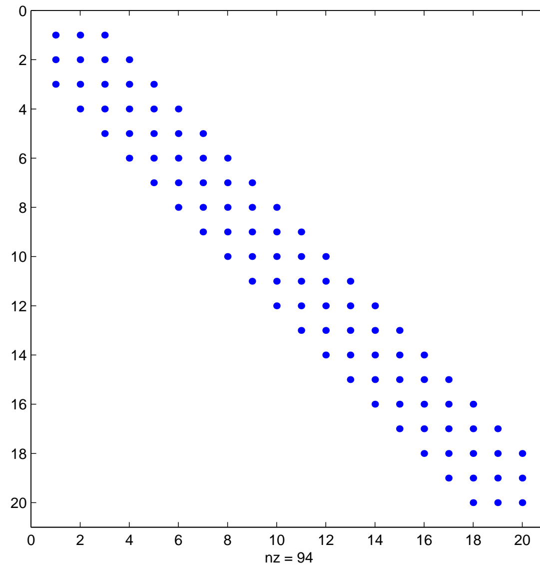
The sparsity pattern of the Hessian matrix has been predetermined and stored in the file `tbroyhstr.mat`. The sparsity structure for the Hessian of this problem is banded, as you can see in the following `spy` plot.

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



In this plot, the center stripe is itself a five-banded matrix. The following plot shows the matrix more clearly:

```
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 requires a huge amount of unnecessary memory and computation. This is because `fmincon` attempts 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 execute `fmincon` as shown in Step 2.

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(fun,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" on page 4-7, and `g` is the gradient.

Because of the five-banded center stripe, you can improve the solution by using a five-banded preconditioner instead of the default diagonal preconditioner. Using the `optimset` function, reset the `PrecondBandWidth` parameter to `2` 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(fun,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 as in Eq. 2-7, 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 long 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 that this information is available in `brownfgh`, using the `GradObj` and `Hessian` options.

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(fun,xstart,[],[],Aeq,beq,[],[],[],options);
```

Setting the option `PrecondBandWidth` to `inf` causes 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
```

Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints

The `fmincon` and `fminunc` large-scale methods can solve problems where the Hessian is dense but structured. For these problems, `fmincon` and `fminunc` do not compute $H*Y$ with the Hessian H directly, as they do for medium-scale problems and for large-scale problems with sparse H, because forming H would be memory-intensive. Instead, you must provide `fmincon` or `fminunc` with a function that, given a matrix Y and information about H, computes $W = H*Y$.

In this example, the objective function is nonlinear and linear equalities exist so `fmincon` is used. The objective function has the structure

$$f(x) = \hat{f}(x) - \frac{1}{2}x^T V V^T x$$

where V is a 1000-by-2 matrix. The Hessian of f is dense, but the Hessian of $\hat{f}$ is sparse. If the Hessian of $\hat{f}$ is $\hat{H}$, then H, the Hessian of f, is

$$H = \hat{H} - VV^T$$

To avoid excessive memory usage that could happen by working with H directly, the example provides a Hessian multiply function, `hmfleq1`. This function, when passed a matrix `Y`, uses sparse matrices `Hinfo`, which corresponds to $\hat{H}$, and `V` to compute the Hessian matrix product

```
W = H*Y = (Hinfo - V*V')*Y
```

In this example, the Hessian multiply function needs $\hat{H}$ and `V` to compute the Hessian matrix product. `V` is a constant, so you can capture `V` in a function handle to an anonymous function.

However, $\hat{H}$ is not a constant and must be computed at the current `x`. You can do this by computing $\hat{H}$ in the objective function and returning $\hat{H}$ as `Hinfo` in the third output argument. By using `optimset` to set the `'Hessian'` options to `'on'`, `fmincon` knows to get the `Hinfo` value from the objective function and pass it to the Hessian multiply function `hmfleq1`.

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

The example passes `brownvv` to `fmincon` as the objective function. The `brownvv.m` file is long and is not included here. You can view the code with the command

```
type brownvv
```

Because `brownvv` computes the gradient and part of the Hessian as well as the objective function, the example (Step 3) uses `optimset` to set the `GradObj` and `Hessian` options to `'on'`.

Step 2: Write a function to compute Hessian-matrix products for H given a matrix Y.

Now, define a function `hmfleq1` that uses `Hinfo`, which is computed in `brownvv`, and `V`, which you can capture in a function handle to an anonymous function, to compute the Hessian matrix product `W` where
`W = H*Y = (Hinfo - V*V')*Y`. This function must have the form

```
W = hmfleq1(Hinfo,Y)
```

The first argument must be the same as the third argument returned by the objective function `brownvv`. The second argument to the Hessian multiply function is the matrix `Y` (of `W = H*Y`).

Because `fmincon` expects the second argument `Y` to be used to form the Hessian matrix product, `Y` is always a matrix with `n` rows where `n` is the number of dimensions in the problem. The number of columns in `Y` can vary. Finally, you can use a function handle to an anonymous function to capture V, so V can be the third argument to `'hmfleqq'`.

```
function W = hmfleq1(Hinfo,Y,V);
%HMFLEQ1 Hessian-matrix product function for BROWNVV objective.
%   W = hmfleq1(Hinfo,Y,V) computes W = (Hinfo-V*V')*Y
%   where Hinfo is a sparse matrix computed by BROWNVV
%   and V is a 2 column matrix.
W = Hinfo*Y - V*(V'*Y);
```

Note The function `hmfleq1` is available in the Optimization Toolbox as the M-file `hmfleq1.m`.

Step 3: Call a nonlinear minimization routine with a starting point and linear equality constraints.

Load the problem parameter, `V`, and the sparse equality constraint matrices, `Aeq` and `beq`, from `fleq1.mat`, which is available in the Optimization Toolbox. Use `optimset` to set the `GradObj` and `Hessian` options to `'on'` and to set the `HessMult` option to a function handle that points to `hmfleq1`. Call `fmincon` with objective function `brownvv` and with `V` as an additional parameter:

```
function [fval, exitflag, output, x] = runfleq1
% RUNFLEQ1 demonstrates 'HessMult' option for FMINCON with linear
```

```
% equalities.

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 1.3.6.2 $  $Date: 2004/02/11 14:43:07 $

problem = load('fleq1');   % Get V, Aeq, beq
V = problem.V; Aeq = problem.Aeq; beq = problem.beq;
n = 1000;             % problem dimension
xstart = -ones(n,1); xstart(2:2:n,1) = ones(length(2:2:n),1); %
starting point
options = optimset('GradObj','on','Hessian','on','HessMult',...
@(Hinfo,Y)hmfleq1(Hinfo,Y,V) ,'Display','iter','TolFun',1e-9);
[x,fval,exitflag,output] =
fmincon(@(x)brownvv(x,V),xstart,[],[],Aeq,beq,[],[],[],options);
```

Note Type `[fval,exitflag,output] = runfleq1` to run the preceding code. This command displays the values for `fval`, `exitflag`, and `output`, as well as the following iterative display.

Because the iterative display was set using `optimset`, the results displayed are

```
                               Norm of      First-order
 Iteration    f(x)          step          optimality  CG-iterations
     1        1997.07             1           555           0
     2        1072.56       6.31716           377           1
     3        480.232       8.19554           159           2
     4        136.861       10.3015          59.5           2
     5        44.3708       9.04697          16.3           2
     6        44.3708           100          16.3           2
     7        44.3708            25          16.3           0
     8       -8.90967          6.25          28.5           0
     9       -318.486          12.5           107           1
    10       -318.486          12.5           107           1
    11       -415.445         3.125          73.9           0
    12       -561.688         3.125          47.4           2
    13       -785.326          6.25           126           3
    14       -785.326       4.30584           126           5
    15       -804.414       1.07646          26.9           0
    16       -822.399       2.16965           2.8           3
```

```
  17      -823.173      0.40754          1.34          3
  18      -823.241     0.154885         0.555          3
  19      -823.246    0.0518407         0.214          5
  2       -823.246   0.00977601       0.00724          6
Optimization terminated successfully:
Relative function value changing by less than OPTIONS.TolFun
```

Convergence is rapid for a problem of this size with the PCG iteration cost increasing modestly as the optimization progresses. Feasibility of the equality constraints is maintained at the solution

```
norm(Aeq*x-beq) =
           1.2861e-013
```

Preconditioning

In this example, `fmincon` cannot use `H` to compute a preconditioner because `H` only exists implicitly. Instead of `H`, `fmincon` uses `Hinfo`, the third argument returned by `brownvv`, to compute a preconditioner. `Hinfo` is a good choice because it is the same size as `H` and approximates `H` to some degree. If `Hinfo` were not the same size as `H`, `fmincon` would compute a preconditioner based on some diagonal scaling matrices determined from the algorithm. Typically, this would not perform as well.

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.

Step 1: Load the Hessian and define f, lb, ub.

```
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;
```

Step 2: Call a quadratic minimization routine with a starting point xstart.

```
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 option 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 tradeoff is beneficial, as the time for `quadprog` to solve the problem decreases by a factor of 10.

Quadratic Minimization with a Dense but Structured Hessian

The `quadprog` large-scale method can also solve large problems where the Hessian is dense but structured. For these problems, `quadprog` does not compute $H*Y$ with the Hessian H directly, as it does for medium-scale problems and for large-scale problems with sparse H, because forming H would be memory-intensive. Instead, you must provide `quadprog` with a function that, given a matrix Y and information about H, computes $W = H*Y$.

In this example, the Hessian matrix `H` has the structure `H = B + A*A'` where `B` is a sparse 512-by-512 symmetric matrix, and `A` is a 512-by-10 sparse matrix composed of a number of dense columns. To avoid excessive memory usage that could happen by working with `H` directly because `H` is dense, the example provides a Hessian multiply function, `qpbox4mult`. This function, when passed a matrix `Y`, uses sparse matrices `A` and `B` to compute the Hessian matrix product `W = H*Y = (B + A*A )*Y`.

In this example, the matrices `A` and `B` need to be provided to the Hessian multiply function `qpbox4mult`. You can pass one matrix as the first argument to `quadprog`, which is passed to the Hessian multiply function. You can use a nested function to provide the value of the second matrix.

Step 1: Decide what part of H to pass to quadprog as the first argument.

Either `A`, or `B` can be passed as the first argument to `quadprog`. The example chooses to pass `B` as the first argument because this results in a better preconditioner (see “Preconditioning” on page 2-68).

```
quadprog(B,f,[],[],[],[],l,u,xstart,options)
```

Step 2: Write a function to compute Hessian-matrix products for H.

Now, define a function rungpbox4t that

- Contains a nested function qpbox4mult that uses A and B to compute the Hessian matrix product W where W = H*Y = (B + A*A)*Y. The nested function must have the form

  ```
  W = qpbox4mult(Hinfo,Y,...)
  ```

  The first two arguments Hinfo and Y are required.

- Loads the problem parameters from qpbox4.mat.
- Uses optimset to set the HessMult option to a function handle that points to qpbox4mult.
- Calls quadprog with B as the first argument.

The first argument to the nested function qpbox4mult must be the same as the first argument passed to quadprog, which in this case is the matrix B.

The second argument to qpbox4mult is the matrix Y (of W = H*Y). Because quadprog expects Y to be used to form the Hessian matrix product, Y is always a matrix with n rows where n is the number of dimensions in the problem. The number of columns in Y can vary. The function qpbox4mult is nested so that the value of the matrix A comes from the outer function.

```
function [fval, exitflag, output, x] = runqpbox4
% RUNQPBOX4 demonstrates 'HessMult' option for QUADPROG with
% bounds.

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 1.5.6.1 $  $Date: 2004/02/11 14:43:46 $

problem = load('qpbox4'); % Get xstart, u, l, B, A, f
xstart = problem.xstart; u = problem.u; l = problem.l;
B = problem.B; A = problem.A; f = problem.f;
mtxmpy = @qpbox4mult; % function handle to qpbox4mult nested
% subfunction

% Choose the HessMult option
options = optimset('HessMult',mtxmpy);
```

```
% Pass B to qpbox4mult via the H argument. Also, B will be used in
% computing a preconditioner for PCG.
% A is passed as an additional argument after 'options'
[x, fval, exitflag, output] = ...
quadprog(B,f,[],[],[],[],l,u,xstart,options);

    function W = qpbox4mult(B,Y);
        %QPBOX4MULT Hessian matrix product with dense structured
        %Hessian.
        %   W = qpbox4mult(B,Y) computes W = (B + A*A')*Y where
        %   INPUT:
        %       B - sparse square matrix (512 by 512)
        %      Y - vector (or matrix) to be multiplied by B + A'*A.
        %   VARIABLES from outer function runqpbox4:
        %       A - sparse matrix with 512 rows and 10 columns.
        %
        %   OUTPUT:
        %       W - The product (B + A*A')*Y.
        %

        % Order multiplies to avoid forming A*A',
        %   which is large and dense
        W = B*Y + A*(A'*Y);
    end

end
```

Step 3: Call a quadratic minimization routine with a starting point.

To call the quadratic minimizing routine contained in `runqpbox4`, enter

```
[fval,exitflag,output] = runqpbox4
```

to run the preceding code and display the values for `fval`, `exitflag`, and `output`. The results are

```
Optimization terminated: relative function value changing by less
 than sqrt(OPTIONS.TolFun), no negative curvature detected in
current
 trust region model and the rate of progress (change in f(x)) is
slow.
```

```
fval =

 -1.0538e+003


exitflag =

     3


output =

       iterations: 18
        algorithm: 'large-scale: reflective trust-region'
    firstorderopt: 0.0028
     cgiterations: 50
          message: [1x206 char]
```

After 18 iterations with a total of 30 PCG iterations, the function value is reduced to

```
fval =
          -1.0538e+003
```

and the first-order optimality is

```
output.firstorderopt =
           0.0043
```

Preconditioning

In this example, `quadprog` cannot use `H` to compute a preconditioner because `H` only exists implicitly. Instead, `quadprog` uses `B`, the argument passed in instead of `H`, to compute a preconditioner. `B` is a good choice because it is the same size as `H` and approximates `H` to some degree. If `B` were not the same size as `H`, `quadprog` would compute a preconditioner based on some diagonal scaling matrices determined from the algorithm. Typically, this would not perform as well.

Because the preconditioner is more approximate than when `H` is available explicitly, adjusting the `TolPcg` parameter to a somewhat smaller value might

be required. This example is the same as the previous one, but reduces TolPcg from the default 0.1 to 0.01.

```
function [fval, exitflag, output, x] = runqpbox4prec
% RUNQPBOX4PREC demonstrates 'HessMult' option for QUADPROG with
bounds.

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 1.5.6.1 $  $Date: 2004/02/11 14:43:46 $

problem = load('qpbox4'); % Get xstart, u, l, B, A, f
xstart = problem.xstart; u = problem.u; l = problem.l;
B = problem.B; A = problem.A; f = problem.f;
mtxmpy = @qpbox4mult; % function handle to qpbox4mult nested
subfunction

% Choose the HessMult option
% Override the TolPCG option
options = optimset('HessMult',mtxmpy,'TolPcg',0.01);

% Pass B to qpbox4mult via the H argument. Also, B will be used in
% computing a preconditioner for PCG.
% A is passed as an additional argument after 'options'
[x, fval, exitflag, output] =
quadprog(B,f,[],[],[],[],l,u,xstart,options);

    function W = qpbox4mult(B,Y);
       %QPBOX4MULT Hessian matrix product with dense structured
        %Hessian.
        %   W = qpbox4mult(B,Y) computes W = (B + A*A')*Y where
        %   INPUT:
        %       B - sparse square matrix (512 by 512)
       %       Y - vector (or matrix) to be multiplied by B + A'*A.
        %   VARIABLES from outer function runqpbox4:
        %       A - sparse matrix with 512 rows and 10 columns.
        %
        %   OUTPUT:
        %       W - The product (B + A*A')*Y.
        %
```

```
            % Order multiplies to avoid forming A*A',
            %   which is large and dense
            W = B*Y + A*(A'*Y);
        end

    end
```

Now, enter

```
[fval,exitflag,output] = runqpbox4prec
```

to run the preceding code. After 18 iterations and 50 PCG iterations, the function value has the same value to five significant digits

```
fval =
-1.0538e+003
```

but the first-order optimality is further reduced.

```
output.firstorderopt =
0.0028
```

Note Decreasing `TolPcg` too much can substantially increase the number of PCG iterations.

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
```

For bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient.

You can improve (decrease) the first-order optimality by using a sparse QR factorization in each iteration. To do this, 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'));
```

Because the iterative display was set using `optimset`, the results displayed 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

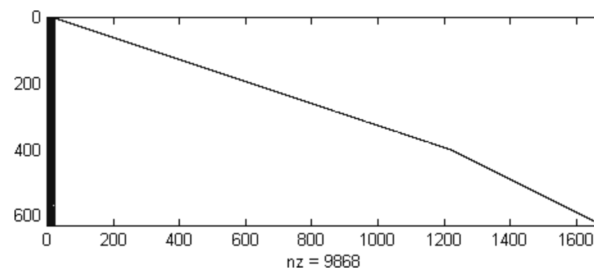
$$\min f^T x \qquad \text{such that} \qquad \begin{aligned} Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

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'));
```

Because the iterative display was set using `optimset`, the results displayed 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 the "Main Algorithm" section in "Large-Scale Linear Programming" on page 4-13.

Default Options Settings

The options structure contains options 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 options is generated. Some of the default options values are calculated using factors based on problem size, such as MaxFunEvals. Some options are dependent on the specific optimization routines and are documented on those function reference pages (See "Function Reference" on page 5-1).

Table , Optimization Options, on page 5-9 provides an overview of all the options in the options structure.

Changing the Default Settings

The function optimset creates or updates an options structure to pass to the various optimization functions. The arguments to the optimset function are option name and option 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 option name uniquely. Case is ignored for option names. For option values that are strings, however, case and the exact string are necessary.

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

Here are some examples of the use of optimset.

Returning All Options

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

Determining Options Used by a Function

The options structure defines the options that can be used by the functions provided by the toolbox. Because functions do not use all the options, it can be useful to find which options 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, enter

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

This command sets the value of the `Display` option to `'iter'`, which causes the toolbox to display output at each iteration. You can also turn off any output display (`'off'`), display output only at termination (`'final'`), or display output only if the problem fails to converge (`'notify'`).

Running Medium-Scale Optimization

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

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

For `fsolve`, the default is the medium-scale algorithm. To use the large-scale algorithm, enter

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

Setting More Than One Option

You can specify multiple options 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 Option Values

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

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

Displaying Iterative Output

This section describes the column headings used in the iterative output of

- Medium-scale algorithms
- Large-scale algorithms

Output Headings: Medium-Scale Algorithms

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

fminsearch

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`.

fzero and fminbnd

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 an interval containing a zero), `bisection`

(bisection search), and `interpolation`. For `fminbnd`, the possible operations are `initial`, `golden` (golden section search), and `parabolic` (parabolic interpolation).

fminunc

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.

lsqnonlin and lsqcurvefit

For `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 "Least-Squares Optimization" on page 3-17. (This value is displayed when you use the Levenberg-Marquardt method and omitted when you use the Gauss-Newton method.)

fsolve

For `fsolve` with the default trust-region dogleg method, the headings are

```
                                 Norm of   First-order Trust-region
 Iteration  Func-count  f(x)      step     optimality       radius
```

where

- `Iteration` is the iteration number.
- `Func-count` is the number of function evaluations.
- `f(x)` is the sum of squares of 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.
- `Trust-region radius` is the radius of the trust region for that step.

For `fsolve` with either the Levenberg-Marquardt or Gauss-Newton method, the headings are

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

where

- `Residual` is the residual (sum of squares) of the function.
- `Step-size` is the step-size in the current search direction.
- `Directional derivative` is the gradient of the function along the search direction.

fmincon and fseminf

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.
- `Procedure` gives a message about the Hessian update and QP subproblem.

The `Procedure` messages are discussed in "Updating the Hessian Matrix" on page 3-30.

For `fgoalattain` and `fminimax`, the headings are the same as for `fmincon` except that `f(x)` and `max constraint` are combined into `Max{F,constraints}`. `Max{F,constraints}` 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

fminunc

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" on page 4-5) at the current (optimization) iteration.

lsqnonlin, lsqcurvefit, and fsolve

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" on page 4-7 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" on page 4-5) at the current (optimization) iteration.

fmincon

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" on page 4-7 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" on page 4-5) at the current (optimization) iteration.

linprog

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}+s-ub` 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" on page 4-13) 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 "Large-Scale Linear Programming" on page 4-13.

Calling an Output Function Iteratively

For some problems, you might need output from an optimization algorithm at each iteration. For example, you might want to find the sequence of points that the algorithm computes and plot those points. To get this information, you can create an output function that the optimization function calls at each iteration. This section provides an example that shows how to do this.

The example in this section continues the one described in “Nonlinear Inequality Constrained Example” on page 2-11, which uses the function `fmincon` to solve a nonlinear, constrained optimization. To run the example, you must first create an M-file for the objective function, `objfcn.m`, and an M-file for the constraints, `confcn.m`, as described in that section.

At each iteration in this example, the output function

- Plots the current point computed by the algorithm.
- Stores the point, its corresponding objective function value, and the current search direction. The search direction is a vector that points in the direction from the current point to the next one.

When the algorithm is complete, the output function saves this information to the MATLAB workspace where you can view it.

Creating the Output Function

To create the output function for the example,

1 Open a new M-file in the MATLAB editor.

2 Copy and paste the following code into the M-file.

```
function stop = outfun(x,optimValues,state)
stop=[];
persistent history
persistent searchdir
hold on
switch state
    case 'init'
        history = []; searchdir = [];
    case 'iter'
```

```
            % Concatenate current point and objective function value
            % with history. x must be a row vector.
            history = [history;[x optimValues.fval]];
            % Concatenate current search direction with searchdir.
            searchdir = [searchdir; optimValues.searchdirection']
            plot(x(1),x(2),'o');
            % Label points with iteration number.
            text(x(1)+.15,x(2),num2str(optimValues.iteration));
        case 'done'
            assignin('base','hist', history);
            assignin('base','search', searchdir);
    otherwise
    end
    hold off
```

3 Save the file as `outfun.m` in a directory on the MATLAB path.

The input arguments that the optimization function passes to `myfunction` are

- `x` — The point computed by the algorithm at the current iteration

  The example keeps a record of these points in the matrix `history` and plots the points.

- `optimValues` — Structure containing data from the current iteration

  The example uses the following fields of `optimValues`.

  - `optimValues.iteration`, which is the number of the current iteration, is the label of the current point in the plot.
  - `optimValues.fval` is the current objective function value in `history`.
  - `optimValues.searchdirection` is the current search direction in `searchdir`.

- `state` — The current state of the algorithm.

  The example determines the current state of the algorithm from `state` and performs tasks accordingly. In this example, `state` has one of the following values at each iteration:

  - `'init'` — The algorithm has not yet started the first iteration.
  - `'iter'` — The algorithm has just completed an iteration.
  - `'done'` — The algorithm has completed the last iteration.

The output argument stop, which this example does not use, returns a flag that tells whether the optimization should quit or continue. You can use stop to modify the criteria that fmincon uses to decide when to halt.

For more information about these arguments, see "Output Function" on page 5-15.

Running the Example

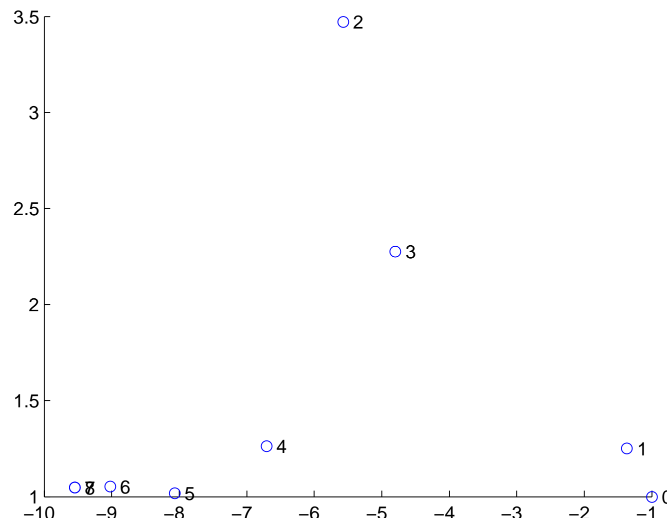
To make the function fmincon call the output function outfun at each iteration, set the options field Outputfcn to @outfun with the following command:

```
options = optimset('OutputFcn',@outfun,'LargeScale','off');
```

Then, to run the example, call fmincon with options as an input argument and using the initial point [-1 1]:

```
x0 = [-1 1];
fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options)
```

This returns a plot of the sequence of points computed by fmincon.



The optimal point occurs at the eighth iteration. Note that the last two points in the sequence are so close that they overlap.

The example returns the sequence of points algorithm computes as a matrix hist and the sequence of search directions as a matrix search. You can view the sequence of points by entering `hist`, which displays the sequence of points in the first two columns and their corresponding objective function values in the third.

```
hist =

   -1.0000    1.0000    1.8394
   -1.3679    1.2500    1.8513
   -5.5708    3.4699    0.3002
   -4.8000    2.2752    0.5298
   -6.7054    1.2618    0.1870
   -8.0679    1.0186    0.0729
   -9.0230    1.0532    0.0353
   -9.5471    1.0471    0.0236
   -9.5474    1.0474    0.0236
```

You can view the sequence of search directions by entering `search`.

```
search =

   -0.3679    0.2500
   -4.2029    2.2199
    0.7708   -1.1947
   -3.8108   -2.0268
   -1.3625   -0.2432
   -0.9552    0.0346
   -0.5241   -0.0061
   -0.0003    0.0003
```

You can see that the search directions point from the current point in the sequence to the next point by computing the differences between consecutive points:

```
hist(2:end,1:2) -  hist(1:end-1,1:2)

ans =
   -0.3679    0.2500
   -4.2029    2.2199
```

```
 0.7708   -1.1947
-1.9054   -1.0134
-1.3625   -0.2432
-0.9552    0.0346
-0.5241   -0.0061
-0.0003    0.0003
```

Optimizing Anonymous Functions Instead of M-Files

The routines in the Optimization Toolbox also perform optimization on anonymous functions, as well as functions defined by M-files.

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

```
fh = @(x)1./((x-0.3).^2 + 0.01) + 1./((x-0.9).^2 + 0.04)-6;
```

The constructor for an anonymous function returns a function handle, shown as `fh` above, that you can use in calling the function. Use the usual MATLAB function regular calling syntax to call a function by means of its function handle.

Evaluate the anonymous function at 2.0:

```
fh(2.0)
ans =
   -4.8552
```

You can also pass handle `fh` to an optimization routine to minimize it:

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

You can create anonymous functions of more than one argument. For example, to use `lsqcurvefit`, you first need a function that takes two input arguments, `x` and `xdata`,

```
fh = @(x,xdata)sin(x).*xdata +(x.^2).*cos(xdata);
x = pi; xdata = pi*[4;2;3];
fh(x, xdata)

ans =

    9.8696
    9.8696
   -9.8696
```

and you then call `lsqcurvefit`.

```
% Assume ydata exists
x = lsqcurvefit(fh,x,xdata,ydata)
```

Other examples that use this technique:

- A matrix equation

  ```
  x = fsolve(@(x)x*x*x-[1,2;3,4],ones(2,2))
  ```

- A nonlinear least-squares problem

  ```
  x = lsqnonlin(@(x)x*x-[3 5;9 10],eye(2,2))
  ```

- An 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`,

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

  solves the problem described on the `fgoalattain` reference page.

Typical Problems and How to Deal with Them

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. You can generally use solutions from lower order problems 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," you can overcome many of the limitations associated with optimization techniques. Additionally, you can handle problems that are not typically in the standard form by using an appropriate transformation. Below is a list of typical problems and recommendations for dealing with them.

Table 2-1: Troubleshooting

Problem	Recommendation
The solution does not appear to be a global minimum.	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 can help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.

Table 2-1: Troubleshooting (Continued)

Problem	Recommendation
`fminunc` produces warning messages and seems to exhibit slow convergence near the solution.	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 might increase the accuracy of gradient calculations.
Sometimes an optimization problem has values of `x` for which it is impossible to evaluate the objective function `fun` or the nonlinear constraints function `nonlcon`.	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.
The function that is being minimized has discontinuities.	The derivation of the underlying method is based upon functions with continuous first and second derivatives. Some success might 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, you can adjust the finite-difference parameters 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, Δx, is always in the range `DiffMinChange < Dx < DiffMaxChange`.

Table 2-1: Troubleshooting (Continued)

Problem	Recommendation
Warning messages are displayed.	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.
The independent variables, x, can only take on discrete values, for example, integers.	This type of problem commonly occurs 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, you can solve some discrete problems by first solving an equivalent continuous problem. Do this by progressively eliminating discrete variables 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.

Table 2-1: Troubleshooting (Continued)

Problem	Recommendation
The minimization routine appears to enter an infinite loop or returns a solution that does not satisfy the problem constraints.	Your objective (`fun`), constraint (`nonlcon, seminfcon`), or gradient (computed by `fun`) functions might be returning `Inf`, `NaN`, or complex values. The minimization routines expect only real numbers to be returned. Any other values can cause unexpected results. Insert some checking code into the user-supplied functions to verify that only real numbers are returned (use the function `isfinite`).
You do not get the convergence you expect from the `lsqnonlin` routine.	You might 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.

Selected Bibliography

[1] Hairer, E., S. P. Norsett, and G. Wanner, *Solving Ordinary Differential Equations I – Nonstiff Problems*, Springer-Verlag, pp. 183-184.

3

Standard Algorithms

Standard Algorithms provides an introduction to the different optimization problem formulations, and describes the *medium-scale* (i.e., standard) algorithms used in the toolbox functions. These algorithms 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.

This chapter consists of these sections:

Optimization Overview (p. 3-3)	Introduces optimization as a way of finding a set of parameters that can in some way be defined as optimal. These parameters are obtained by minimizing or maximizing an objective function, subject to equality or inequality constraints and/or parameter bounds.
Unconstrained Optimization (p. 3-4)	Discusses the use of quasi-Newton and line search methods for unconstrained optimization.
Quasi-Newton Implementation (p. 3-10)	Provides implementation details for the Hessian update and line search phases of the quasi-Newton algorithm.
Least-Squares Optimization (p. 3-17)	Discusses the use of the Gauss-Newton and Levenberg-Marquardt methods for nonlinear least-squares (LS) optimization. Also provides implementation details for the Gauss-Newton and Levenberg-Marquardt methods used in the nonlinear least-squares optimization routines, `lsqnonlin` and `lsqcurvefit`.
Nonlinear Systems of Equations (p. 3-23)	Discusses the use of Gauss-Newton, Newton's, and trust-region dogleg methods for the solution of nonlinear systems of equations. Also provides implementation details for the Gauss-Newton and trust-region dogleg methods used by the `fsolve` function.

Constrained Optimization (p. 3-27)	Discusses the use of the Kuhn-Tucker (KT) equations as the basis for sequential quadratic programming (SQP) methods. Provides implementation details for the Hessian matrix update, quadratic programming problem solution, and line search and merit function calculation phases of the SQP algorithm used in `fmincon`, `fminimax`, `fgoalattain`, and `fseminf`. Explains the simplex algorithm, which is an optional algorithm for `linprog`.
Multiobjective Optimization (p. 3-41)	Introduces multiobjective optimization and discusses strategies for dealing with competing objectives. It discusses in detail the use of the goal attainment method, and suggests improvements to the SQP method for use with the goal attainment method.
Selected Bibliography (p. 3-51)	Lists published materials that support concepts implemented in the medium-scale algorithms.

Note Medium-scale is not a standard term and is used here only to differentiate these algorithms from the large-scale algorithms described in "Large-Scale Algorithms" on page 4-1.

Optimization Overview

Optimization techniques are 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 might 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, might 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

$$\underset{x}{\text{minimize}} \quad f(x) \tag{3-1}$$

subject to

$$\begin{aligned} G_i(x) &= 0, & i &= 1, \ldots, m_e \\ G_i(x) &\le 0, & i &= m_e + 1, \ldots, m \end{aligned}$$

where *x* is the vector of length *n* design parameters, *f(x)* is the objective function, which returns a scalar value, and the vector function *G(x)* returns a vector of length *m* containing the values of the equality and inequality constraints evaluated at *x* .

An efficient and accurate solution to this problem depends not only 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 (LP) problem. 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 can 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 subproblem.

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 [32]) 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, because 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

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{3-2}$$

The minimum of this function is at $x = [1,1]$ where $f(x) = 0$. A contour map of this function is shown in Figure 3-1, 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 zigzagging from one side of the valley to another. Note that toward the center of the plot, a number of larger steps are taken when a point lands exactly at the center of the valley.

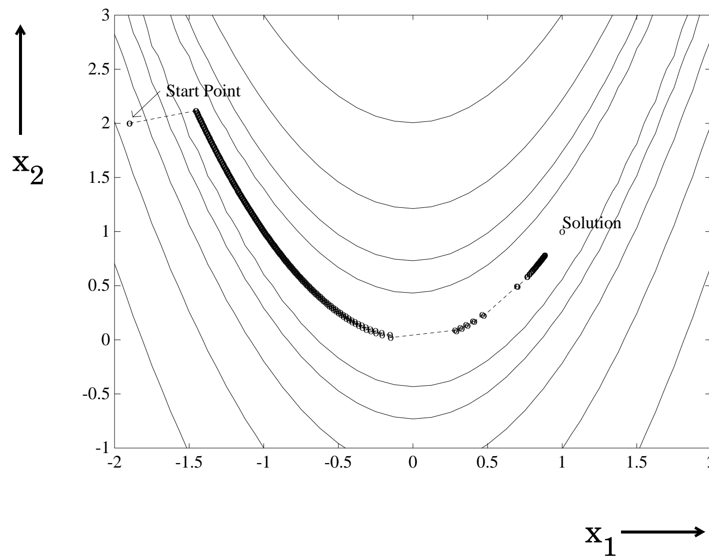


Figure 3-1: Steepest Descent Method on Rosenbrock's Function (Eq. 2-2)

This type of function (Eq. 3-2), also known as the banana function, is notorious in unconstrained examples because of the way the curvature bends around the origin. Eq. 3-2 is used throughout this section to illustrate the use of a variety of optimization techniques. The contours have been plotted in exponential increments because of the steepness of the slope surrounding the U-shaped valley.

This section continues with discussions of the following:

- Quasi-Newton Methods
- Line Search
- Quasi-Newton Implementation

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

$$\min_{x} \frac{1}{2}x^T H x + c^T x + b \qquad \textbf{(3-3)}$$

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.,

$$\nabla f(x^*) = Hx^* + c = 0 \qquad \textbf{(3-4)}$$

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

$$x^* = -H^{-1}c \qquad \textbf{(3-5)}$$

Newton-type methods (as opposed to quasi-Newton methods) calculate H directly and proceed in a direction of descent 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. However, the formula of Broyden [3], Fletcher [14], Goldfarb [22], and Shanno [39] (BFGS) is thought to be the most effective for use in a General Purpose method.

The formula given by BFGS is

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T s_k^T s_k H_k}{s_k^T H_k s_k} \qquad \textbf{(3-6)}$$

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 that avoids the direct inversion of H 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 [9], Fletcher, and Powell [16]. This uses the same formula as the BFGS method (Eq. 3-6) 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

$$d = -H_k^{-1} \cdot \nabla f(x_k) \tag{3-7}$$

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function (Eq. 3-2) in Figure 3-2, BFGS Method on Rosenbrock's Function. 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.

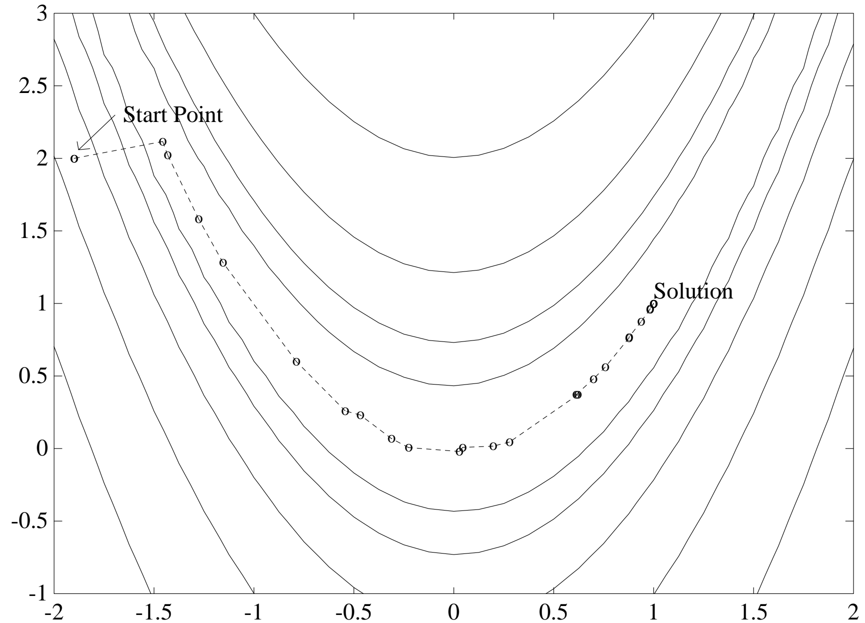


Figure 3-2: BFGS Method on Rosenbrock's Function

Line Search

Line search is a search method that is used as part of a larger optimization algorithm. At each step of the main algorithm, the line-search method searches along the line containing the current point, x_k, parallel to the *search direction*, which is a vector determined by the main algorithm. That is, the method finds the next iterate x_{k+1} of the form

$$x_{k+1} = x_k + \alpha^* d_k \quad \textbf{(3-8)}$$

where x_k denotes the current iterate, d_k is the search direction, and α is a scalar step-length parameter.

The line search method attempts to decrease the objective function along the line $x_k + \alpha d$ by repeatedly minimizing polynomial interpolation models of the objective function. The line search procedure has two main steps:

- The *bracketing* phase determines the range of points on the line $x_{k+1} = x_k + \alpha^* d_k$ to be searched. The *bracket* corresponds to an interval specifying the range of values of α.
- The *sectioning* step divides the bracket into subintervals, on which the minimum of the objective function is approximated by polynomial interpolation.

The resulting step length α satisfies the Wolfe conditions:

$$f(x_k + \alpha d_k) \le f(x_k) + c_1 \alpha \nabla f_k^T d_k \qquad \textbf{(3-9)}$$

$$\nabla f(x_k + \alpha d_k)^T d_k \ge c_2 \alpha \nabla f_k^T d_k \qquad \textbf{(3-10)}$$

where c_1 and c_2 are constants with $0 < c_1 < c_2 < 1$.

The first condition (3-1) requires that α_k gives sufficiently decreases the objective function. The second condition(3-2) ensures that the step length is not too small.

The line search method is an implementation of the algorithm described in Section 2-6 of [15]. See also [33] for more information about line search.

Quasi-Newton Implementation

This section describes the implementation of the quasi-Newton method in the toolbox. The algorithm consists of two phases:

- Determination of a direction of search (Hessian update)
- Line search procedures

Hessian Update

Many of the optimization functions determine the direction of search by updating the Hessian matrix at each iteration, using the BFGS method (Eq. 3-6). The function `fminunc` also provides an option to use the DFP method given in "Quasi-Newton Methods" on page 3-6 (set `HessUpdate` to `'dfp'` in `options` 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. You achieve positive definiteness of H by ensuring that H is initialized to be positive definite and thereafter $q_k^T s_k$ (from Eq. 3-11) 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,

$$q_k^T s_k = \alpha_k (\nabla f(x_{k+1})^T d - \nabla f(x_k)^T d) \tag{3-11}$$

You always achieve the condition that $q_k^T s_k$ is positive by performing a sufficiently accurate line search. 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

After choosing the direction of the search, the optimization function uses a line search procedure to determine how far to move in the search direction. This section describes the line search procedures used by the functions `lsqnonlin`, `lsqcurvefit`, and `fsolve`.

The functions use one of two line search strategies, 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 cubic and quadratic polynomial method.

Cubic Polynomial Method

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

$$f(x_{k+1}) < f(x_k) \qquad \textbf{(3-12)}$$

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

$$x_{k+1} = x_k + \alpha_k d \qquad \textbf{(3-13)}$$

If this step does not satisfy the condition (Eq. 3-12), 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. 3-12), 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.

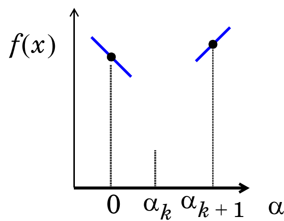
It is usual practice to reset α_k to unity after every iteration. However, note that the quadratic model (Eq. 3-3) is generally only a good one near to the solution point. Therefore, α_k is modified at each major iteration to compensate for the case when the approximation to the Hessian is monotonically 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.

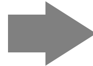
Cubic Polynomial Line Search Procedures. After each update procedure, a step length α_k is attempted, following which a number of scenarios are possible. Consideration of all the possible cases is quite complicated and so they are represented pictorially below.

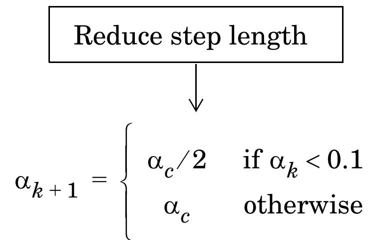
For each case:

- The left point on the graph 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 point.
- The right point is the point x_{k+1} after a step of α_k is taken in the direction d.

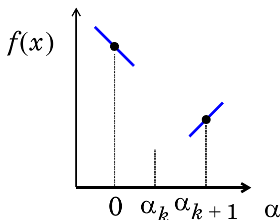
Case 1: $f(x_{k+1}) > f(x_k)$, $\nabla f(x_{k+1})^T d > 0$

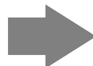


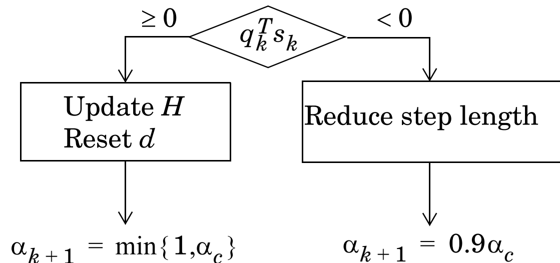




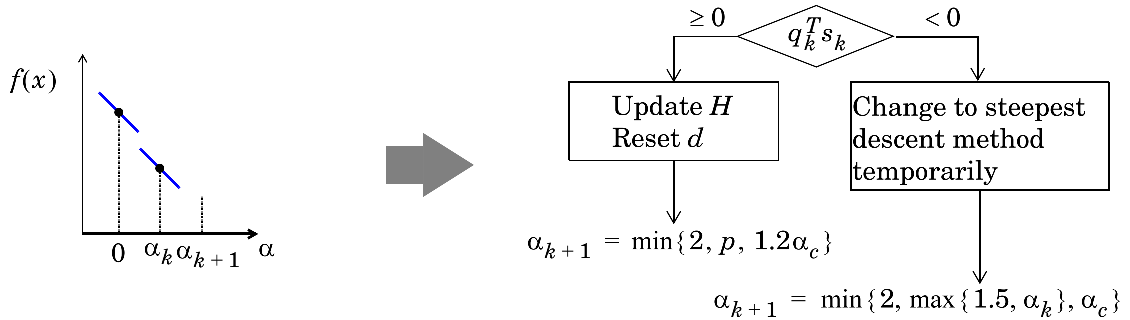
Case 2: $f(x_{k+1}) \le f(x_k)$, $\nabla f(x_{k+1})^T d \ge 0$



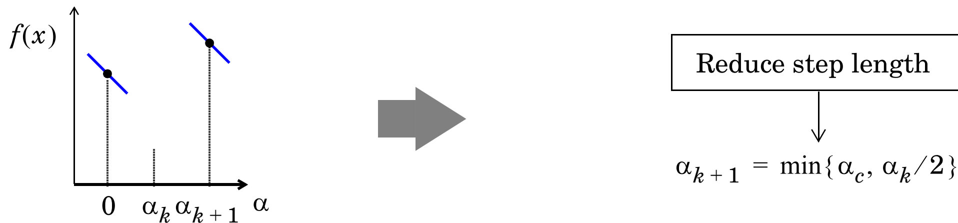




Case 3: $f(x_{k+1}) < f(x_k),\ \nabla f(x_{k+1})^T d < 0$



Case 4: $f(x_{k+1}) \geq f(x_k),\ \nabla f(x_{k+1})^T d \leq 0$ where $p = 1 + q_k^T s_k - \nabla f(x_{k+1})^T d + \min\{0, \alpha_{k+1}\}$



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 can be negative, in which case α_c is given a value of $2\alpha_k$.

Certain robustness measures have also been included so that, even in the case when false gradient information is supplied, you can achieve a reduction in *f(x)* by taking a negative step. You do this 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 and 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, 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 `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 the following graphs for a number of point combinations. The left-most point in each graph represents the function value $f(x_1)$ and univariate gradient $\nabla f(x_k)$ obtained at the last update. The remaining points represent the points accumulated in the minor iterations of the line search procedure.

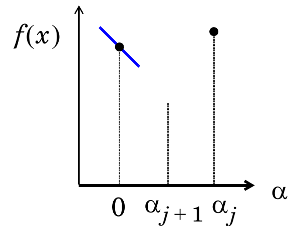
The terms α_q and α_c refer to the minimum obtained from a respective quadratic and cubic interpolation or extrapolation. For highly nonlinear functions, α_c and α_q can be negative, in which case they are set to a value of $2\alpha_k$ so that they are always maintained to be positive. Cases 1 and 2 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.

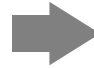
When the minimum is finally bracketed, cubic interpolation is achieved using one gradient and three function evaluations. If the interpolated point is greater than any of the three used for the interpolation, then it is replaced with the

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.

The following graphs illustrate the line search procedures for cases 1 through 4, with a gradient only for the first point.

Case 1: $f(x_j) \geq f(x_k)$

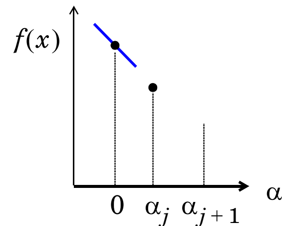


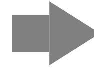


Reduce step length

$\alpha_{j+1} = \alpha_q$

Case 2: $f(x_j) < f(x_k)$

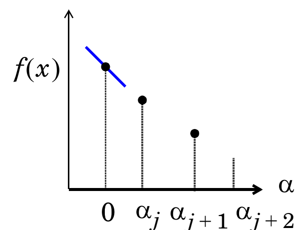


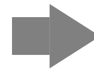


Increase step length

$\alpha_{j+1} = 1.2\alpha_q$

Case 3: $f(x_{j+1}) < f(x_k)$

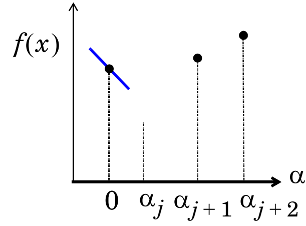


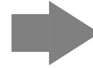


Increase step length

$\alpha_{j+2} = \max\{1.2\alpha_q, 2\alpha_{j+1}\}$

Case 4: $f(x_{j+1}) > f(x_k)$





Reduce step length

$\alpha_{j+2} = \alpha_c$

Least-Squares Optimization

The line search procedures used in conjunction with a quasi-Newton method are used as part of the 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.

$$\min_{x \in \Re^n} f(x) = \frac{1}{2}\| F(x) \|_2^2 = \frac{1}{2}\sum_i F_i(x)^2 \qquad \textbf{(3-14)}$$

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

$$\min_{x \in \Re^n} \int_{t_2}^{t_1} (y(x,t) - \phi(t))^2 dt \qquad \textbf{(3-15)}$$

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

When the integral is discretized using a suitable quadrature formula, Eq. 3-15 can be formulated as a least-squares problem:

$$\min_{x \in \Re^n} f(x) = \sum_{i=1}^{m} (\bar{y}(x,t_i) - \bar{\phi}(t_i))^2 \qquad \textbf{(3-16)}$$

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

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

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. 3-15) can be minimized using a

general unconstrained minimization technique, as described in "Unconstrained Optimization" on page 3-4, 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. 3-15) have a special structure.

Denoting the m-by-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 $H_i(x)$, you have

$$
\begin{aligned}
G(x) &= 2J(x)^T F(x) \\
H(x) &= 2J(x)^T J(x) + 2Q(x)
\end{aligned}
\tag{3-17}
$$

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.

This section continues with discussions of the following:

- Gauss-Newton Method
- Levenberg-Marquardt Method
- Nonlinear Least-Squares Implementation

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.

$$
\min_{x \in \Re^n} \| J(x_k) d_k - F(x_k) \|_2^2
\tag{3-18}
$$

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.

Consider the efficiencies that are possible with the Gauss-Newton method. Figure 3-3 shows the path to the minimum on Rosenbrock's function (Eq. 3-2) 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.

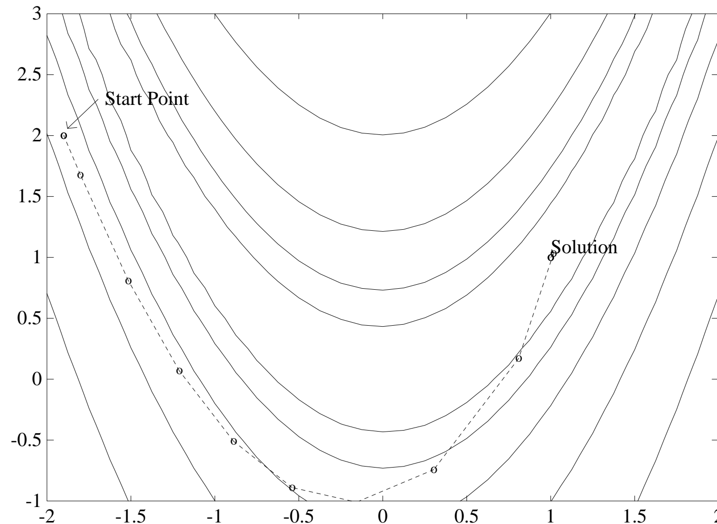


Figure 3-3: Gauss-Newton Method on Rosenbrock's Function

The Gauss-Newton method often encounters problems when the second order term *Q(x)* in Eq. 3-17 is significant. A method that overcomes this problem is the Levenberg-Marquardt method.

Levenberg-Marquardt Method

The Levenberg-Marquardt [27],[29] method uses a search direction that is a solution of the linear set of equations

$$(J(x_k)^T J(x_k) + \lambda_k I) d_k = -J(x_k) F(x_k) \quad \textbf{(3-19)}$$

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 toward 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 3-4, Levenberg-Marquardt Method on Rosenbrock's Function. The solution for Rosenbrock's function (Eq. 3-2) 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 the increased robustness of the Levenberg-Marquardt method compensates for its occasional poorer efficiency.

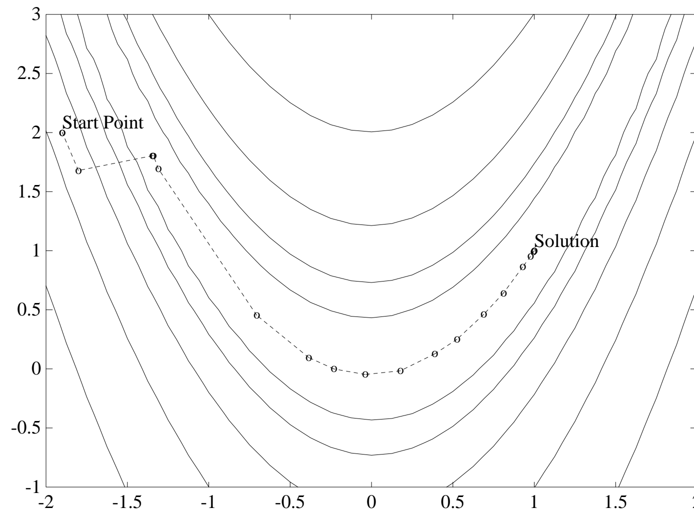


Figure 3-4: Levenberg-Marquardt Method on Rosenbrock's Function

Nonlinear Least-Squares Implementation

For a general survey of nonlinear least-squares methods, see Dennis [10]. Specific details on the Levenberg-Marquardt method can be found in Moré [30]. 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
- Levenberg-Marquardt Implementation

Gauss-Newton Implementation

The Gauss-Newton method is implemented using polynomial line search strategies similar to those discussed for unconstrained optimization. In solving the linear least-squares problem (Eq. 3-15), you can avoid exacerbation of the conditioning of the equations 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 (1e-15 in this implementation) 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

$$f_p(x_k) = J(x_{k-1})d_{k-1} + F(x_{k-1}) \tag{3-20}$$

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 the following figure.

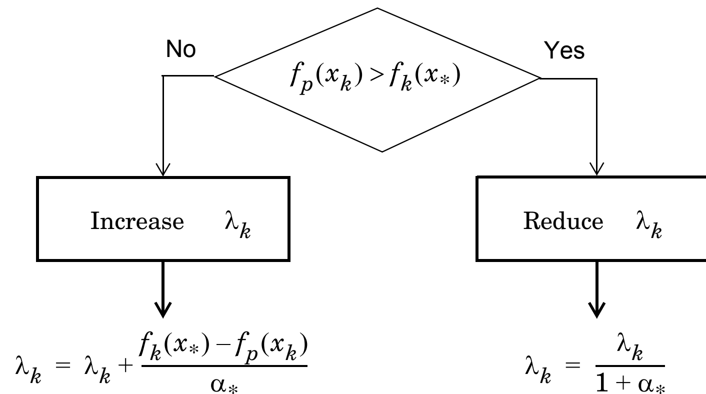


Figure 3-5: Updating λ_k

Following the update of λ_k, a solution of Eq. 3-19 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`. You can select the Gauss-Newton method by setting `LevenbergMarquardt` to `'off'` in `options`.

Nonlinear Systems of Equations

Solving a nonlinear system of equations $F(x)$ involves finding a solution such that every equation in the nonlinear system is 0. That is, we have n equations and n unknowns and we want to find $x \in \Re^n$ such that $F(x) = 0$ where

$$F(x) = \begin{bmatrix} F_1(x) \\ F_2(x) \\ \vdots \\ F_n(x) \end{bmatrix}$$

The assumption is that a zero, or root, of the system exists. These equations may represent economic constraints, for example, that must all be satisfied.

Gauss-Newton Method

One approach to solving this problem is to use a Nonlinear Least-Squares solver, such those described in “Least-Squares Optimization” on page 3-17. Since we assume the system has a root, it would have a small residual, and so using the Gauss-Newton Method is effective. In this case, at each iteration we solve a linear least-squares problem, as described in Eq. 3-18, to find the search direction. (See “Gauss-Newton Method” on page 3-18 for more information.)

Trust-Region Dogleg Method

Another approach is to solve a linear system of equations to find the search direction, namely, Newton’s method says to solve for the search direction d_k such that

$$J(x_k)d_k = -F(x_k)$$
$$x_{k+1} = x_k + d_k$$

where $J(x_k)$ is the n-by-n Jacobian

$$J(x_k) = \begin{bmatrix} \nabla F_1(x_k)^T \\ \nabla F_2(x_k)^T \\ \vdots \\ \nabla F_n(x_k)^T \end{bmatrix}$$

Newton's method can run into difficulties. $J(x_k)$ may be singular, and so the Newton step d_k is not even defined. Also, the exact Newton step d_k may be expensive to compute. In addition, Newton's method may not converge if the starting point is far from the solution.

Using trust-region techniques (introduced in "Trust-Region Methods for Nonlinear Minimization" on page 4-2) improves robustness when starting far from the solution and handles the case when $J(x_k)$ is singular. To use a trust-region strategy, a merit function is needed to decide if x_{k+1} is better or worse than x_k. A possible choice is

$$\min_d f(d) = \frac{1}{2} F(x_k + d)^T F(x_k + d)$$

But a minimum of $f(d)$ is not necessarily a root of $F(x)$.

The Newton step d_k is a root of

$$M(x_k + d) = F(x_k) + J(x_k)d$$

and so it is also a minimum of $m(d)$ where

$$\begin{aligned} \min_d m(d) = \frac{1}{2} \| M(x_k + d) \|_2^2 &= \frac{1}{2} \| F(x_k) + J(x_k)d \| \\ &= \frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) \\ &\quad + \frac{1}{2} d^T (J(x_k)^T J(x_k))\, d \end{aligned} \quad \textbf{(3-21)}$$

Then $m(d)$ is a better choice of merit function than $f(d)$, and so the trust region subproblem is

$$\min_d \left[\frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T (J(x_k)^T J(x_k))\, d \right] \quad \textbf{(3-22)}$$

such that $\| D \cdot d \| \leq \Delta$. This subproblem can be efficiently solved using a dogleg strategy.

For an overview of trust-region methods, see Conn [5], and Nocedal [33].

Nonlinear Equations Implementation

Both the Gauss-Newton and trust-region dogleg methods are implemented in the Optimization Toolbox. Details of their implementations are discussed below.

Gauss-Newton Implementation

The Gauss-Newton implementation is the same as that for least-squares optimization. It is described in "Gauss-Newton Implementation" on page 3-21.

Trust-Region Dogleg Implementation

The key feature of this algorithm is the use of the Powell dogleg procedure for computing the step d, which minimizes Eq. 3-22. For a detailed description, see Powell [36].

The step d is constructed from a convex combination of a Cauchy step (a step along the steepest descent direction) and a Gauss-Newton step for $f(x)$. The Cauchy step is calculated as

$$d_C = -\alpha J(x_k)^T F(x_k)$$

where α is chosen to minimize Eq. 3-21.

The Gauss-Newton step is calculated by solving

$$J(x_k) \cdot d_{GN} = -F(x_k)$$

using the MATLAB \ (matrix left division) operator.

The step d is chosen so that

$$d = d_C + \lambda(d_{GN} - d_C)$$

where λ is the largest value in the interval [0,1] such that $\| d \| \leq \Delta$. If J_k is (nearly) singular, d is just the Cauchy direction.

The dogleg algorithm is efficient since it requires only one linear solve per iteration (for the computation of the Gauss-Newton step). Additionally, it can be more robust than using the Gauss-Newton method with a line search.

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 that 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. 3-1), the Kuhn-Tucker equations can be stated as

$$\begin{aligned} \nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \cdot \nabla G_i(x^*) &= 0 \\ \lambda_i^* \cdot G_i(x^*) &= 0 \qquad i = 1, \ldots, m \\ \lambda_i^* &\geq 0 \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{3-23}$$

in addition to the original constraints in Equation 3-1.

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. Because 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. 3-23.

The solution of the KT equations forms the basis to many nonlinear programming algorithms. These algorithms attempt to compute the Lagrange multipliers directly. 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 subproblem is solved at each major iteration (also known as Iterative Quadratic Programming, Recursive Quadratic Programming, and Constrained Variable Metric methods).

This section continues with discussions of the following topics:

- "Sequential Quadratic Programming (SQP)" on page 3-28
- A "Quadratic Programming (QP) Subproblem" on page 3-29
- "SQP Implementation" on page 3-30
- "Simplex Algorithm" on page 3-36

Sequential Quadratic Programming (SQP)

SQP methods represent the state of the art in nonlinear programming methods. Schittkowski [38], for example, has implemented and tested a version that outperforms 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 [1], Han [24], and Powell ([34],[35]), 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 subproblem whose solution is used to form a search direction for a line search procedure. An overview of SQP is found in Fletcher [15], Gill et al. [21], Powell [37], and Schittkowski [25]. The general method, however, is stated here.

Given the problem description in GP (Eq. 3-1) the principal idea is the formulation of a QP subproblem based on a quadratic approximation of the Lagrangian function.

$$L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g_i(x) \qquad \textbf{(3-24)}$$

Here you simplify Eq. 3-1 by assuming that bound constraints have been expressed as inequality constraints. You obtain the QP subproblem by linearizing the nonlinear constraints.

Quadratic Programming (QP) Subproblem

$$\begin{aligned} &\underset{d \in \Re^n}{\text{minimize}} \quad \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d \\ &\nabla g_i(x_k)^T d + g_i(x_k) = 0 \qquad i = 1, \dots m_e \\ &\nabla g_i(x_k)^T d + g_i(x_k) \le 0 \qquad i = m_e + 1, \dots m \end{aligned} \tag{3-25}$$

This subproblem can be solved using any QP algorithm (see, for instance, “Quadratic Programming Solution” on page 3-32). 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 “Updating the Hessian Matrix” on page 3-30). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function (Eq. 3-24). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see “Updating the Hessian Matrix” on page 3-30) 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 informed decisions regarding directions of search and step length.

Consider Rosenbrock’s function (Eq. 3-2) with an additional nonlinear inequality constraint, *g(x)*,

$$x_1^2 + x_2^2 - 1.5 \le 0 \tag{3-26}$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. Figure 3-6 shows the path to the solution point $x = [0.9072, 0.8228]$ starting at $x = [-1.9, 2]$.

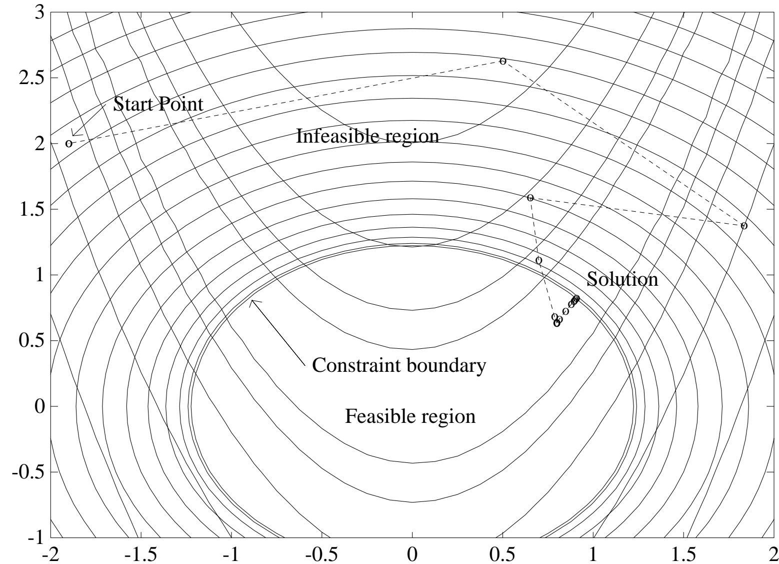


Figure 3-6: SQP Method on Nonlinear Linearly Constrained Rosenbrock's Function (Eq.3-2)

SQP Implementation

The SQP implementation consists of three main stages, which are discussed briefly in the following subsections:

- 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 $(i = 1, \ldots, m)$ is an estimate of the Lagrange multipliers.

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T H_k}{s_k^T H_k s_k} \text{ where} \qquad \textbf{(3-27)}$$

$$s_k = x_{k+1} - x_k$$

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

Powell [35] recommends keeping the Hessian positive definite even though it might 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 $q_k.*\dot{s}_k$ is repeatedly halved. This procedure is continued until $q_k^T s_k$ is greater than or equal to 1*e*-5. If, after this procedure, $q_k^T s_k$ is still not positive, modify q_k by adding a vector v multiplied by a constant scalar w, that is,

$$q_k = q_k + wv \qquad \textbf{(3-28)}$$

where

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

$$\text{if } (q_k)_i \cdot w < 0 \text{ and } (q_k)_i \cdot (s_k)_i < 0 (i = 1, \ldots m)$$

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

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

The functions `fmincon`, `fminimax`, `fgoalattain`, and `fseminf` all use SQP. If `Display` is set to `'iter'` in `options`, 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 preceding procedure 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 subproblem is infeasible,

then `infeasible` is displayed. Such displays are usually not a cause for concern but indicate that the problem is highly nonlinear and that convergence might 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 of the following form is solved, where A_i refers to the `i`th row of the m-by-n matrix A.

$$\begin{aligned} \underset{d \in \Re^n}{\text{minimize}} \quad & q(d) = \frac{1}{2} d^T H d + c^T d \\ & A_i d = b_i \qquad i = 1, \dots, m_e \\ & A_i d \le b_i \qquad i = m_e + 1, \dots, m \end{aligned} \tag{3-29}$$

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 [20] and [19]. It has been modified for both Linear Programming (LP) and Quadratic Programming (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, $\bar{A}_k$, is maintained that is an estimate of the active constraints (i.e., those that 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 $\bar{A}_k^T$, where l is the number of active constraints and $l < m$. That is, Z_k is given by

$$Z_k = Q[:, l+1:m] \tag{3-30}$$

where

$$Q^T \bar{A}_k^T = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

Once Z_k is found, 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 you view the quadratic as a function of p, by substituting for $\hat{d}_k$, you have

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p \tag{3-31}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c \tag{3-32}$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c \tag{3-33}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k \qquad \text{where } \hat{d}_k = Z_k^T p \tag{3-34}$$

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 $\bar{A}_k$. If such a step can be taken, without violation of the constraints, then this is the solution to QP (Eq. 3-30). 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 iteration. The distance to the constraint boundaries in any direction $\hat{d}_k$ is given by

$$\alpha = \min_i \left\{ \frac{-(A_i x_k - b_i)}{A_i \hat{d}_k} \right\} \qquad (i = 1, \ldots, m) \tag{3-35}$$

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

$$\bar{A}_k^T \lambda_k = c \tag{3-36}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Eq. 3-30). However, if any component of λ_k is negative, and the component 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 you can find a point by solving the linear programming problem

$$\begin{aligned} &\underset{\gamma \in \Re,\, x \in \Re^n}{\text{minimize}} \quad \gamma \\ &A_i x = b_i \qquad i = 1, \ldots, m_e \\ &A_i x - \gamma \le b_i \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{3-37}$$

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Eq. 3-37 by setting x to a value that satisfies the equality constraints. You can determine this value by solving an under- or overdetermined 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.

You can modify the preceding QP algorithm 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).

$$\hat{d}_k = -Z_k Z_k^T g_k \tag{3-38}$$

If a feasible point is found using the preceding 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

$$H\hat{d}_1 = -g_k \tag{3-39}$$

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 subproblem produces a vector d_k , which is used to form a new iterate

$$x_{k+1} = x_k + \alpha d_k \tag{3-40}$$

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 [24] and Powell [35] of the following form is used in this implementation.

$$\Psi(x) = f(x) + \sum_{i=1}^{m_e} r_i \cdot g_i(x) + \sum_{i=m_e+1}^{m} r_i \cdot \max\{0, g_i(x)\} \tag{3-41}$$

Powell recommends setting the penalty parameter

$$r_i = (r_{k+1})_i = \max_i\left\{\lambda_i, \frac{1}{2}((r_k)_i + \lambda_i)\right\}, \qquad i = 1, \ldots, m \tag{3-42}$$

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

$$r_i = \frac{\| \nabla f(x) \|}{\| \nabla g_i(x) \|} \quad (3\text{-}43)$$

where $\| \ \|$ 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.

Simplex Algorithm

The simplex algorithm, invented by George Dantzig in 1947, is one of the earliest and best known optimization algorithms. The algorithm solves the linear programming problem

$$\min_x f^T x \quad \text{subject to} \quad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

The algorithm moves along the edges of the polyhedron defined by the constraints, from one vertex to another, while decreasing the value of the objective function, $f^T x$, at each step. This section describes an improved version of the original simplex algorithm that returns a vertex optimal solution.

This section covers the following topics:

- "Main Algorithm" on page 3-37
- "Preprocessing" on page 3-38
- "Using the Simplex Algorithm" on page 3-38
- "Basic and Nonbasic Variables" on page 3-39
- "References" on page 3-40

Main Algorithm

The simplex algorithm has two phases:

- Phase 1 — Compute an initial basic feasible point.
- Phase 2 — Compute the optimal solution to the original problem.

Note You cannot supply an initial point `x0` for `linprog` with the simplex algorithm. If you pass in `x0` as an input argument, `linprog` ignores `x0` and computes its own initial point for the algorithm.

Phase 1. In phase 1, the algorithm finds an initial basic feasible solution (see "Basic and Nonbasic Variables" on page 3-39 for a definition) by solving an auxiliary piecewise linear programming problem. The objective function of the auxiliary problem is the *linear penalty function* $P = \sum_j P_j(x_j)$, where $P_j(x_j)$ is defined by

$$P_j(x_j) = \begin{cases} x_j - u_j & \text{if} \quad x_j > u_j \\ 0 & \text{if} \; l_j \le x_j \le u_j \\ l_j - x_j & \text{if} \quad l_j > x_j \end{cases}$$

$P(x)$ measures how much a point x violates the lower and upper bound conditions. The auxiliary problem is

$$\min_x \sum_j P_j \quad \text{subject to} \quad \begin{array}{l} A \cdot x \le b \\ Aeq \cdot x = beq \end{array}$$

The original problem has a feasible basis point if and only if the auxiliary problem has minimum value 0.

The algorithm finds an initial point for the auxiliary problem by a heuristic method that adds slack and artificial variables as necessary. The algorithm then uses this initial point together with the simplex algorithm to solve the auxiliary problem. The optimal solution is the initial point for phase 2 of the main algorithm.

Phase 2. In phase 2, the algorithm applies the simplex algorithm, starting at the initial point from phase 1, to solve the original problem. At each iteration, the algorithm tests the optimality condition and stops if the current solution is optimal. If the current solution is not optimal, the algorithm

1. Chooses one variable, called the *entering variable*, from the nonbasic variables and adds the corresponding column of the nonbasis to the basis (see "Basic and Nonbasic Variables" on page 3-39 for definitions).

2. Chooses a variable, called the *leaving variable*, from the basic variables and removes the corresponding column from the basis.

3. Updates the current solution and the current objective value.

The algorithm chooses the entering and the leaving variables by solving two linear systems while maintaining the feasibility of the solution.

Preprocessing

The simplex algorithm uses the same preprocessing steps as the large-scale linear programming solver, which are described in "Preprocessing" on page 4-16. In addition, the algorithm uses two other steps:

1. Eliminates columns that have only one nonzero element and eliminates their corresponding rows.

2. For each constraint equation $a \cdot x = b$, where a is a row of Aeq, the algorithm computes the lower and upper bounds of the linear combination $a \cdot x$ as rlb and rub if the lower and upper bounds are finite. If either rlb or rub equals b, the constraint is called a *forcing constraint*. The algorithm sets each variable corresponding to a nonzero coefficient of $a \cdot x$ equal its upper or lower bound, depending on the forcing constraint. The algorithm then deletes the columns corresponding to these variables and deletes the rows corresponding to the forcing constraints.

Using the Simplex Algorithm

To use the simplex method, set `'LargeScale'` to `'off'` and `'Simplex'` to `'on'` in `options`.

```
options = optimset('LargeScale', 'off', 'Simplex', 'on')
```

Then call the function `linprog` with the `options` input argument. See the reference page for `linprog` for more information.

`linprog` returns empty output arguments for `x` and `fval` if it detects infeasibility or unboundedness in the preprocessing procedure. `linprog` returns the current point when it

- Exceeds the maximum number of iterations
- Detects that the problem is infeasible or unbounded in phases 1 or 2

When the problem is unbounded, `linprog` returns `x` and `fval` in the unbounded direction.

Basic and Nonbasic Variables

This section defines the terms *basis*, *nonbasis*, and *basic feasible solutions* for a linear programming problem. The definition assumes that the problem is given in the following standard form:

$$\min_x f^T x \quad \text{such that} \quad \begin{array}{l} A \cdot x = b \\ lb \le x \le ub \end{array}$$

(Note that A and b are not the matrix and vector defining the inequalities in the original problem.) Assume that A is an m-by-n matrix, of rank $m < n$, whose columns are $\{a_1, a_2, ..., a_n\}$. Suppose that $\{a_{i_1}, a_{i_2}, \ldots, a_{i_m}\}$ is a basis for the column space of A, with index set $B = \{i_1, i_2, ..., i_m\}$, and that $N = \{1, 2, ..., n\} \backslash B$ is the complement of B. The submatrix A_B is called a *basis* and the complementary submatrix A_N is called a *nonbasis*. The vector of *basic variables* is x_B and the vector of *nonbasic variables* is x_N. At each iteration in phase 2, the algorithm replaces one column of the current basis with a column of the nonbasis and updates the variables x_B and x_N accordingly.

If x is a solution to $A \cdot x = b$ and all the nonbasic variables in x_N are equal to either their lower or upper bounds, x is called a *basic solution*. If, in addition, the basic variables in x_B satisfy their lower and upper bounds, so that x is a feasible point, x is called a *basic feasible solution*.

References

[1] Chvatal, Vasek, *Linear Programming*, W. H. Freeman and Company, 1983.

[2] Bixby, Robert E., "Implementing the Simplex Method: The Initial Basis," ORSA Journal on Computing, Vol. 4, No. 3, 1992.

[3] Andersen, Erling D. and Knud D. Andersen, "Presolving in Linear Programming," Mathematical Programming, Vol. 71, pp. 221-245, 1995.

Multiobjective Optimization

The rigidity of the mathematical problem posed by the general optimization formulation given in GP (Eq. 3-1) 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 tradeoffs between the objectives fully understood. As the number of objectives increases, tradeoffs 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 to be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

This section includes

- An introduction to multiobjective optimization, which looks at a number of alternative methods
- A discussion of the goal attainment method, which can be posed as a nonlinear programming problem
- Algorithm improvements to the SQP method, for use with the goal attainment method

Introduction

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

$$\begin{aligned}
&\underset{x \in \Re^n}{\text{minimize}} \quad F(x) \\
&\qquad G_i(x) = 0 \qquad i = 1, \ldots, m_e \\
&\qquad G_i(x) \le 0 \qquad i = m_e + 1, \ldots, m \\
&\qquad x_l \le x \le x_u
\end{aligned} \tag{3-44}$$

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 [41] (also called Pareto optimality [4],[6]) 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.,

$$\Omega = \{x \in \Re^n\} \quad \textbf{(3-45)}$$

subject to

$$g_i(x) = 0 \qquad i = 1, \ldots, m_e$$
$$g_i(x) \le 0 \qquad i = m_e + 1, \ldots, m$$
$$x_l \le x \le x_u$$

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

$$\Lambda = \{y \in \Re^m\} \text{ where } y = F(x) \text{ subject to } x \in \Omega \quad \textbf{(3-46)}$$

The performance vector, *F(x)*, maps parameter space into objective function space as is represented for a two-dimensional case in Figure 3-7.

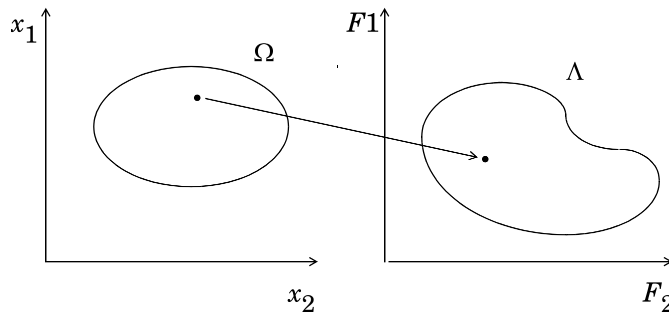


Figure 3-7: Mapping from Parameter Space into Objective Function Space

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

$$F_i(x^* + \Delta x) \le F_i(x^*) \qquad i = 1, \ldots, m$$
$$F_j(x^* + \Delta x) < F_j(x^*) \qquad \text{for some } j \tag{3-47}$$

In the two-dimensional representation of Figure 3-8, Set of Noninferior Solutions, the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.

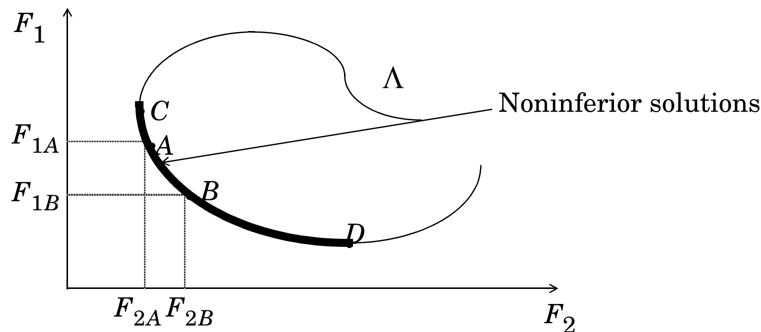


Figure 3-8: Set of Noninferior Solutions

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. Subsequent sections describe some of the techniques.

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.

$$\underset{x \in \Omega}{\text{minimize}} \quad f(x) = \sum_{i=1}^{m} w_i \cdot F_i(x) \tag{3-48}$$

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 tradeoffs between the objectives to be expressed. Further, the noninferior solution boundary can be nonconcurrent, so that certain solutions are not accessible.

This can be illustrated geometrically. Consider the two-objective case in Figure 3-9, Geometrical Representation of the Weighted Sum Method. In the objective function space a line, L, $w^T F(x) = c$ is drawn. The minimization of Eq. 3-48 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.

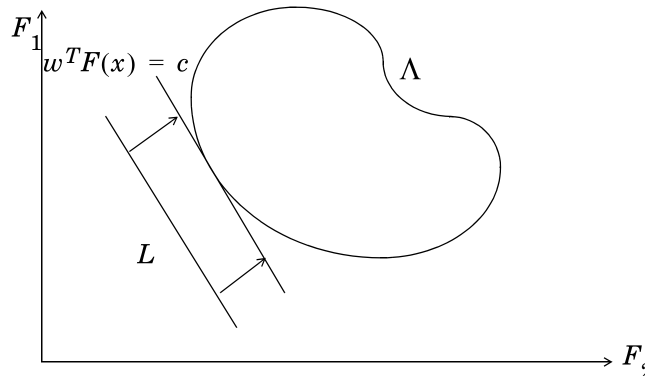


Figure 3-9: Geometrical Representation of the Weighted Sum Method

The aforementioned convexity problem arises when the lower boundary of Λ is nonconvex as shown in Figure 3-10, Nonconvex Solution Boundary. In this case the set of noninferior solutions between A and B is not available.

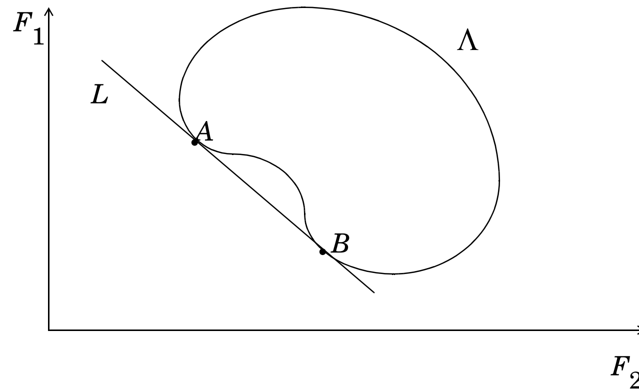


Figure 3-10: Nonconvex Solution Boundary

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

$$\underset{x \in \Omega}{\text{minimize}} \quad F_p(x) \tag{3-49}$$

$$\text{subject to } F_i(x) \le \varepsilon_i \qquad i = 1, \ldots, m \qquad i \neq p$$

Figure 3-11, Geometrical Representation of e-Constraint Method, shows a two-dimensional representation of the ε-constraint method for a two-objective problem.

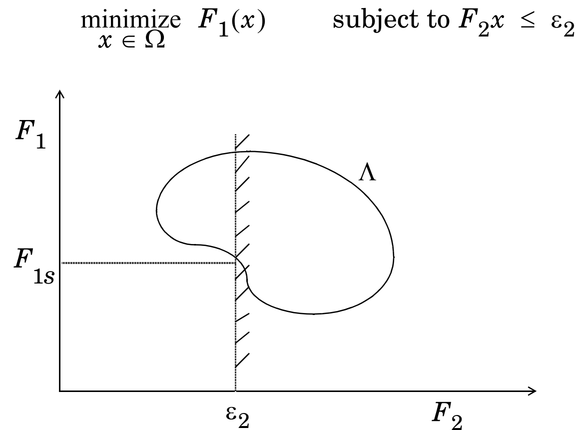


Figure 3-11: Geometrical Representation of ε-Constraint Method

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 [40], that 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 [26]). 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 [18]. 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 overachieved, enabling the designer to be relatively imprecise about initial design goals. The relative degree of under- or overachievement 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.

$$\underset{\gamma \in \Re,\, x \in \Omega}{\text{minimize}} \quad \gamma \tag{3-50}$$

$$\text{such that } F_i(x) - w_i\gamma \le F_i^* \qquad i = 1, \ldots, m$$

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 tradeoffs between the objectives. For instance, setting the weighting vector w equal to the initial goals indicates that the same percentage under- or overattainment of the goals, F^*, is achieved. You can incorporate hard constraints 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 the goal attainment method in control system design can be found in Fleming ([12],[13]).

The goal attainment method is represented geometrically in Figure 3-12, Geometrical Representation of Goal Attainment Method, for the two-dimensional problem.

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

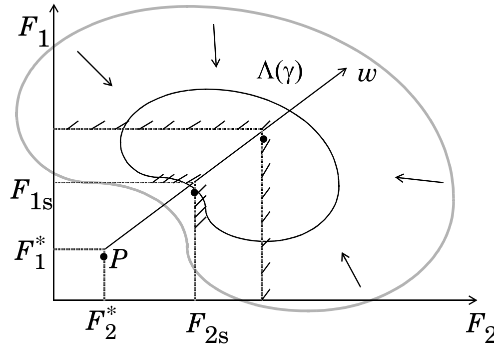


Figure 3-12: Geometrical Representation of Goal Attainment Method

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, Schittkowski [38]). In goal attainment programming there might be a more appropriate merit function, which you can achieve by posing Eq. 3-50 as the minimax problem

$$\underset{x \in \Re^n}{\text{minimize}}\ \max_i\ \{\Lambda_i\} \tag{3-51}$$

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

Following the argument of Brayton et al. [2] for minimax optimization using SQP, using the merit function of Eq. 3-41 for the goal attainment problem of Eq. 3-51 gives

$$\psi(x, \gamma) = \gamma + \sum_{i=1}^{m} r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\} \tag{3-52}$$

When the merit function of Eq. 3-52 is used as the basis of a line search procedure, then, although $\psi(x, \gamma)$ might decrease for a step in a given search direction, the function `max` Λ_i might 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, $\psi(x, \gamma)$ might 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. [2], a solution is therefore to set $\psi(x)$ equal to the worst case objective, i.e.,

$$\psi(x) = \max_i \Lambda_i \tag{3-53}$$

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. 3-53 then becomes infinite for arbitrary violations of the constraints. To overcome this problem while still retaining the features of Eq. 3-53, the merit function is combined with that of Eq. 3-42, giving the following:

$$\psi(x) = \sum_{i=1}^{m} \begin{cases} r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\} & \text{if } w_i = 0 \\ \max_i \Lambda_i, \quad i = 1, \ldots, m & \text{otherwise} \end{cases} \tag{3-54}$$

Another feature that can be exploited in SQP is the objective function γ. From the KT equations (Eq. 3-23) 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. However, this property does not appear if H is initialized as the identity matrix. 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 "Quadratic Programming Solution" on page 3-32.

The preceding modifications have been implemented in `fgoalattain` and have been found to make the method more robust. However, because of 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. 3-41.

Selected Bibliography

[1] 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.

[2] 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.

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

[5] Conn, N.R., N.I.M. Gould, and Ph.L. Toint, *Trust-Region Methods*, MPS/SIAM Series on Optimization, SIAM and MPS, 2000.

[6] 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.

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

[8] Dantzig, G., 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.

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

[10] Dennis, J.E., Jr., "Nonlinear least-squares," *State of the Art in Numerical Analysis* ed. D. Jacobs, Academic Press, pp 269-312, 1977.

[11] Dennis, J.E., Jr. and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall Series in Computational Mathematics, Prentice-Hall, 1983.

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

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

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

[15] Fletcher, R., “Practical Methods of Optimization,” John Wiley and Sons, 1987.

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

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

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

[19] 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.

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

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

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

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

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

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

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

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

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

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

[30] 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.

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

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

[33] Nocedal, J. and S.J. Wright, *Numerical Optimization*, Springer Series in Operations Research, Springer Verlag, 1999.

[34] 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.

[35] 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.

[36] Powell, M.J.D., “A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations,” *Numerical Methods for Nonlinear Algebraic Equations*, (P. Rabinowitz, ed.), Ch.7, 1970.

[37] 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.

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

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

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

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

Large-Scale Algorithms

Large-Scale Algorithms describes the methods used in the Optimization Toolbox to solve large-scale optimization problems. It consists of these sections:

Trust-Region Methods for Nonlinear Minimization (p. 4-2)	Introduces the trust-regions, and describes the use of trust-regions for unconstrained nonlinear minimization.
Preconditioned Conjugate Gradients (p. 4-5)	Presents an algorithm that uses Preconditioned Conjugate Gradients (PCG) for solving large symmetric positive definite systems of linear equations.
Linearly Constrained Problems (p. 4-7)	Discusses the solution of linear equality constrained and box constrained minimization problems.
Nonlinear Least-Squares (p. 4-10)	Describes the solution of nonlinear least-squares problems.
Quadratic Programming (p. 4-11)	Describes the solution of minimization problems with quadratic objective functions.
Linear Least-Squares (p. 4-12)	Describes the solution of linear least-squares problems.
Large-Scale Linear Programming (p. 4-13)	Describes the use of LIPSOL (Linear Interior Point Solver) for the solution of large-scale linear programming problems.
Selected Bibliography (p. 4-17)	Lists published materials that support concepts implemented in the large-scale algorithms.

Trust-Region Methods for Nonlinear Minimization

Many of the methods used in the Optimization Toolbox are based on trust-regions, 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 you are at a point x in n-space and you 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,

$$\min_{s}\{q(s) \quad s \in N\} \tag{4-1}$$

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. This section focuses on the unconstrained problem. Later sections discuss additional complications due to the presence of constraints on the variables.

In the standard trust-region method ([8]), 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

$$\min \quad \left\{\frac{1}{2}s^T Hs + s^T g \quad \text{such that} \quad \|Ds\| \le \Delta\right\} \tag{4-2}$$

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. 4-2 (see [8]); 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. 4-2. 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. 4-2, have been proposed in the literature ([2],[10]). The approximation approach followed in the Optimization Toolbox is to restrict the trust-region subproblem to a two-dimensional subspace S ([1],[2]). Once the subspace S has been computed, the work to solve Eq. 4-2 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

$$H \cdot s_2 = -g \qquad \textbf{(4-3)}$$

or a direction of negative curvature,

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

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. 4-2 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., $f(x+s) \geq f(x)$. See [6],[9] 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 calculate matrix-vector products of the form $H \cdot v$ where v is an arbitrary vector. The symmetric positive definite matrix M is a *preconditioner* for H. That is, $M = C^2$ where $C^{-1}HC^{-1}$ is a well-conditioned matrix or a matrix with clustered eigenvalues.

Algorithm

The Optimization Toolbox uses this PCG algorithm, which it refers to as 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., $d^T H d \leq 0$. 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 "Trust-Region Methods for Nonlinear Minimization" on page 4-2.

Linearly Constrained Problems

Linear constraints complicate the situation described for unconstrained minimization. However, the underlying ideas described previously 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

$$\min \{f(x) \text{ such that } Ax = b\} \tag{4-5}$$

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 A is assumed to be of rank m.

The method used to solve Eq. 4-5 differs from the 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

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{4-6}$$

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., $C \approx H$. See [6] for more details.

Box Constraints

The box constrained problem is of the form

$$\min \{f(x) \text{ such that } l \le x \le u\} \tag{4-7}$$

where l is a vector of lower bounds, and u is a vector of upper bounds. Some (or all) of the components of l can be equal to $-\infty$ and some (or all) of the components of u can 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. 4-7,

$$(D(x))^{-2}g = 0 \tag{4-8}$$

where

$$D(x) = \mathrm{diag}\left(|v_k|^{-\frac{1}{2}}\right)$$

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. 4-8 is not differentiable everywhere. Nondifferentiability occurs when $v_i = 0$. You can avoid such points by maintaining strict feasibility, i.e., restricting $l < x < u$.

The scaled modified Newton step s_k for the nonlinear system of equations given by Eq. 4-8 is defined as the solution to the linear system

$$\hat{M}Ds^N = -\hat{g} \tag{4-9}$$

at the kth iteration, where

$$\hat{g} = D^{-1}g = \mathrm{diag}\left(|v|^{\frac{1}{2}}\right)g \tag{4-10}$$

and

$$\hat{M} = D^{-1}HD^{-1} + \mathrm{diag}(g)J^{v} \qquad \textbf{(4-11)}$$

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 might not be differentiable. $J^{v}_{ii} = 0$ is defined 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

$$f(x) = \frac{1}{2}\sum_i f_i^2(x) = \frac{1}{2}\|F(x)\|_2^2 \tag{4-12}$$

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” on page 4-2. 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

$$\min \quad \|Js + F\|_2^2 \tag{4-13}$$

(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

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

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

$$f(x) = \frac{1}{2}\|Cx + d\|_2^2$$

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.,

$$C^T Cx = -C^T d$$

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

$$\min\ f^T x \quad \text{such that} \begin{pmatrix} Aeq \cdot x = beq \\ Aineq \cdot x \leq bineq \\ l \leq x \leq u \end{pmatrix} \tag{4-14}$$

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.

This section continues with descriptions of

- The main algorithm
- Preprocessing steps

Main Algorithm

The algorithm begins by applying a series of preprocessing steps (see "Preprocessing" on page 4-16). After preprocessing, the problem has the form

$$\min\ f^T x \qquad \text{such that} \begin{pmatrix} A \cdot x = b \\ 0 \leq x \leq u \end{pmatrix} \tag{4-15}$$

The upper bounds constraints are implicitly included in the constraint matrix A. With the addition of primal slack variables s, Eq. 4-15 becomes

$$\min\ f^T x \qquad \text{such that} \begin{pmatrix} A \cdot x = b \\ x + s = u \\ x \geq 0, s \geq 0 \end{pmatrix} \tag{4-16}$$

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

$$\max\ b^T y - u^T w \qquad \text{such that} \qquad \begin{matrix} A^T \cdot y - w + z = f \\ z \geq 0, w \geq 0 \end{matrix} \tag{4-17}$$

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. 4-16 and the dual Eq. 4-17, are

$$F(x, y, z, s, w) = \begin{pmatrix} A \cdot x - b \\ x + s - u \\ A^T \cdot y - w + z - f \\ x_i z_i \\ s_i w_i \end{pmatrix} = 0 \tag{4-18}$$

$$x \geq 0,\ z \geq 0,\ s \geq 0,\ w \geq 0$$

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. 4-18, 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. 4-16.)

The algorithm is a variant of the predictor-corrector algorithm proposed by Mehrotra. Consider an iterate $v = [x;\ y;\ z;\ s;\ w]$, where $[x;\ z;\ s;\ w] > 0$. 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. The two directions are combined 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 preceding steps, 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. 4-18.

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 problem, and not the original. Thus, if the multipliers are *not* requested, this transformation back is not computed, and might save some time computationally.

Selected Bibliography

[1] Branch, M.A., T.F. Coleman, and Y. Li, “A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems,” *SIAM Journal on Scientific Computing*, Vol. 21, Number 1, pp 1-23, 1999.

[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, “A Preconditioned Conjugate Gradient Approach to Linear Equality Constrained Minimization,” submitted to *Computational Optimization and Applications*.

[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.

5

Function Reference

Functions — Categorical List (p. 5-2)	Lists the functions in the toolbox by category.
Function Arguments (p. 5-5)	Describes the input and output arguments of the toolbox functions.
Optimization Options (p. 5-9)	Describes optimization options.
Functions — Alphabetical List (p. 5-25)	Lists the functions in the toolbox alphabetically.

Functions — Categorical List

The Optimization Toolbox provides these categories of functions.

Minimization	Minimization functions
Equation Solving	Solution of linear and nonlinear equations
Least Squares (Curve Fitting)	Linear and nonlinear curve fitting
Utility	Setting and retrieving optimization options
Demos of Large-Scale Methods	Demonstration programs of large-scale methods
Demos of Medium-Scale Methods	Demonstration programs of medium-scale methods

Minimization

`bintprog`	Binary integer programming
`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

`\`	Use \ (left division) to solve linear equations. See the Arithmetic Operators reference page in the online MATLAB documentation.
`fsolve`	Nonlinear equation solving

`fzero`	Scalar nonlinear equation solving

Least Squares (Curve Fitting)

`\`	Use `\` (left division) for linear least squares with no constraints. See the Arithmetic Operators reference page.
`lsqlin`	Constrained linear least squares
`lsqcurvefit`	Nonlinear curve fitting
`lsqnonlin`	Nonlinear least squares
`lsqnonneg`	Nonnegative linear least squares

Utility

`fzmult`	Multiplication with fundamental nullspace basis
`gangstr`	Zero out “small” entries subject to structural rank
`optimget`	Get optimization options values
`optimset`	Create or edit optimization `options` structure

Demos of Large-Scale Methods

From the MATLAB Help browser, click the demo name to run the demo. Look for information and additional instructions in the MATLAB Command Window.

`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

Demos of Medium-Scale Methods

From the MATLAB Help browser, click the demo name to run the demo. Look for information and additional instructions in the MATLAB Command Window.

`bandemo`	Minimization of the `banana` function
`dfildemo`	Finite-precision filter design (requires the Signal Processing Toolbox)
`goaldemo`	Goal attainment example
`optdemo`	Menu of demo routines
`tutdemo`	Script for the medium-scale algorithms. The script follows the "Tutorial" chapter of the Optimization Toolbox User's Guide.

Function Arguments

The Optimization Toolbox functions use these arguments.

Input Arguments	General descriptions of input arguments used by toolbox functions.
Output Arguments	General descriptions of output arguments used by toolbox functions.

Individual function reference pages provide function-specific information, as necessary.

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-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-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 underdetermined linear system and the right-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 is a function handle for an M-file function or a function handle for an anonymous function. See the individual function reference pages for more information on fun.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, lsqnonlin

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. You can also specify unbounded variables using -Inf (for lower bounds) or Inf (for upper bounds). For example, if lb(i) = -Inf, 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. "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the function nonlcon, if necessary. See the individual reference pages for more information on nonlcon.	fgoalattain, fmincon, fminimax
ntheta	The number of semi-infinite constraints.	fseminf
options	An structure that defines options used by the optimization functions. For information about the options, see "Optimization Options" on page 5-9 or the individual function reference pages.	All functions
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. "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize seminfcon, if necessary. See the function reference pages for fseminf for more information on seminfcon.	fseminf

Argument	Description	Used by Functions
`weight`	A weighting vector to control the relative underattainment or overattainment of the objectives.	`fgoalattain`
`xdata`, `ydata`	The input data `xdata` and the observed output data `ydata` that are to be fitted 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`

Output Arguments

Argument	Description	Used by Functions
`attainfactor`	The attainment factor at the solution `x`.	`fgoalattain`
`exitflag`	An integer identifying the reason the optimization algorithm terminated. You can use `exitflag` as a programming tool when writing M-files that perform optimizations. See the reference pages for the optimization functions for descriptions of `exitflag` specific to each function. You can also return a message stating why an optimization terminated by calling the optimization function with the output argument `output` and then displaying `output.message`.	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`

Argument	Description	Used by Functions
grad	The value of the gradient of fun at the solution x. If fun does not compute the gradient, grad is a finite-differencing approximation of the gradient.	fmincon, fminunc
hessian	The value of the Hessian of fun at the solution x. For large-scale methods, if fun does not compute the Hessian, hessian is a finite-differencing approximation of the Hessian. For medium-scale methods, hessian is the value of the Quasi-Newton approximation to the Hessian at the solution x.	fmincon, fminunc
jacobian	The value of the Jacobian of fun at the solution x. If fun does not compute the Jacobian, jacobian is a finite-differencing approximation of the Jacobian.	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
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 of the optimization routine when it terminated prematurely.	All functions

Optimization Options

The following table describes fields in the optimization options structure options. You can set values of these fields using the function optimset. The column labeled L, M, B indicates whether the option applies to large-scale methods, medium scale methods, or both:

- L – Large-scale methods only
- M – Medium-scale methods only
- B – Both large- and medium-scale methods

See the optimset reference page and the individual function reference pages for information about option values and defaults.

The default values for the options vary depending on which optimization function you call with options as an input argument. You can determine determine the default option values for any of the optimization functions by entering optimset followed by the name of the function. For example,

```
optimset fmincon
```

returns a list of the options and the default values for fmincon. Options whose default values listed as [] are not used by the function.

Option Name	Description	L, M, B	Used by Functions
BranchStrategy	Strategy bintprog uses to select branch variable	M	bintprog
DerivativeCheck	Compare user-supplied analytic derivatives (gradients or Jacobian) to finite differencing derivatives.	B	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin
Diagnostics	Display diagnostic information about the function to be minimized or solved.	B	All but fminbnd, fminsearch, fzero, and lsqnonneg

Option 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; `'notify'` displays output only if function does not converge.	B	All. See the individual function reference pages for the values that apply.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no warning.	B	All
`GoalsExactAchieve`	Specifies the number of objectives for which it is required for the objective `fun` to equal the goal `goal`. Such objectives should be partitioned into the first few elements of `F`.	M	`fgoalattain`
`GradConstr`	Gradients for the nonlinear constraints defined by the user.	M	`fgoalattain`, `fmincon`, `fminimax`

Option Name	Description	L, M, B	Used by Functions
GradObj	Gradients for the objective functions defined by the user.	B	fgoalattain, fmincon, fminimax, fminunc, fseminf
Hessian	If 'on', function uses user-defined Hessian or Hessian information (when using HessMult), for the objective function. If 'off', function approximates the Hessian using finite differences.	L	fmincon, fminunc
HessMult	Hessian multiply function defined by the user. "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the Hessian multiply function, if necessary.	L	fmincon, fminunc, quadprog
HessPattern	Sparsity pattern of the Hessian for finite differencing. The size of the matrix is n-by-n, where n is the number of elements in x0, the starting point.	L	fmincon, fminunc
HessUpdate	Quasi-Newton updating scheme.	M	fminunc
InitialHessMatrix	Initial quasi-Newton matrix	M	fminunc
InitialHessType	Initial quasi-Newton matrix type	M	fminunc

Option Name	Description	L, M, B	Used by Functions
`Jacobian`	If `'on'`, function uses user-defined Jacobian or Jacobian information (when using `JacobMult`), for the objective function. If `'off'`, function approximates the Jacobian using finite differences.	B	`fsolve`, `lsqcurvefit`, `lsqnonlin`
`JacobMult`	Jacobian multiply function defined by the user. "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the Jacobian multiply function, if necessary.	L	`fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`
`JacobPattern`	Sparsity pattern of the Jacobian for finite differencing. The size of the matrix is m-by-n, where m is the number of values in the first argument returned by the user-specified function `fun`, and n is the number of elements in `x0`, the starting point.	L	`fsolve`, `lsqcurvefit`, `lsqnonlin`
`LargeScale`	Use large-scale algorithm if possible.	B	`fmincon`, `fminunc`, `fsolve`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`LevenbergMarquardt`	Choose Levenberg-Marquardt over Gauss-Newton algorithm.	M	`lsqcurvefit`, `lsqnonlin`
`LineSearchType`	Line search algorithm choice	M	`fsolve`, `lsqcurvefit`, `lsqnonlin`

Option Name	Description	L, M, B	Used by Functions
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
MaxNodes	Maximum number of possible solutions, or *nodes*, the binary integer programming function bintprog searches	M	bintprog
MaxPCGIter	Maximum number of iterations of preconditioned conjugate gradients method allowed	L	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog
MaxRLPIter	Maximum number of iterations of linear programming relaxation method allowed	M	bintprog
MaxSQPIter	Maximum number of iterations of sequential quadratic programming method allowed	M	fmincon
MaxTime	Maximum amount of time in seconds allowed for the algorithm	M	bintprog
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
NodeDisplayInterval	Node display interval for bintprog	M	bintprog

Option Name	Description	L, M, B	Used by Functions
NodeSearchStrategy	Search strategy that bintprog uses	M	bintprog
NonlEqnAlgorithm	Choose Levenberg-Marquardt or Gauss-Newton over the trust-region dogleg algorithm.	M	fsolve
OutputFcn	Specify a user-defined function that the optimization function calls at each iteration. See "Output Function" on page 5-15.	B	fgoalattain, fmincon, fminimax, fminunc, fseminf, lsqcurvefit, lsqnonlin
PrecondBandWidth	Upper bandwidth of preconditioner for PCG.	L	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog
Simplex	If 'on', function uses the simplex algorithm.	M	linprog
TolCon	Termination tolerance on the constraint violation.	B	bintprog, fgoalattain, fmincon, fminimax, fseminf
TolFun	Termination tolerance on the function value.	B	bintprog, fgoalattain, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, linprog (large-scale only), lsqcurvefit, lsqlin (large-scale only), lsqnonlin, quadprog (large-scale only)
TolPCG	Termination tolerance on the PCG iteration.	L	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog

Option Name	Description	L, M, B	Used by Functions
TolRLPFun	Termination tolerance on the function value of a linear programming relaxation problem	M	bintprog
TolX	Termination tolerance on x.	B	All functions except the medium-scale algorithms for linprog, lsqlin, and quadprog
TolXInteger	Tolerance within which bintprog considers the value of a variable to be an integer	M	bintprog
TypicalX	Typical x values. The length of the vector is equal to the number of elements in x0, the starting point.	B	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog

Output Function

The Outputfcn field of the options structure specifies a function that an optimization function calls at each iteration. Typically, you might use an output function to plot points at each iteration or to display data from the algorithm. To set up an output function, do the following:

1. Write the output function as an M-file function or subfunction.

2. Use optimset to set the value of Outputfcn to be a function handle, that is, the name of the function preceded by the @ sign. For example, if the output function is outfun.m, the command

   ```
   options = optimset('OutputFcn', @outfun);
   ```

   sets the value of OutputFcn to be the handle to outfun.

3. Call the optimization function with options as an input argument.

See "Calling an Output Function Iteratively" on page 2-85 for an example of an output function.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the output function `OutputFcn`, if necessary.

Structure of the Output Function

The function definition line of the output function has the following form:

```
stop = outfun(x, optimValues, state)
```

where

- `x` is the point computed by the algorithm at the current iteration.
- `optimValues` is a structure containing data from the current iteration. "Fields in optimValues" on page 5-16 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 5-23 lists the possible values.
- `stop` is a flag that is `true` or `false` depending on whether the optimization routine should quit or continue. See "Stop Flag" on page 5-23 for more information.

The optimization function passes the values of the input arguments to `outfun` at each iteration.

Fields in optimValues

The following table lists the fields of the `optimValues` structure. A particular optimization function returns values for only some of these fields. For each field, the Returned by Functions column of the table lists the functions that return the field.

Corresponding Output Arguments. Some of the fields of `optimValues` correspond to output arguments of the optimization function. After the final iteration of the optimization algorithm, the value of such a field equals the corresponding output argument. For example, `optimValues.fval` corresponds to the output argument `fval`. So, if you call `fmincon` with an output function and return `fval`, the final value of `optimValues.fval` equals `fval`. The Description column of the following table indicates the fields that have a corresponding output argument.

Command-Line Display. The values of some fields of `optimValues` are displayed at the command line when you call the optimization function with the `Display` field of `options` set to `'iter'`, as described in "Displaying Iterative Output" on page 2-79. For example, `optimValues.fval` is displayed in the `f(x)` column. The Command-Line Display column of the following table indicates the fields that you can display at the command line.

In the following table, the letters L, M, and B mean the following:

- L — Function returns a value to the field when using large-scale algorithm.
- M — Function returns a value to the field when using medium-scale algorithm.
- B — Function returns a value to the field when using both large and medium-scale algorithms.

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`cgiterations`	Number of conjugate gradient iterations at current iteration. Final value equals optimization function output `output.cgiterations`.	`fmincon` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	`CG-iterations` See "Displaying Iterative Output" on page 2-79.
`constrviolation`	Maximum constraint violation	`fgoalattain` (M), `fmincon` (M), `fminimax` (M), `fseminf` (M)	`max constraint` See "Displaying Iterative Output" on page 2-79.

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`degenerate`	Measure of degeneracy. A point is *degenerate* if • The partial derivative with respect to one of the variables is 0 at the point. • A bound constraint is active for that variable at the point. See "Degeneracy" on page 5-22.	`fmincon` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	None
`directionalderivative`	Directional derivative in the search direction	`fgoalattain` (M), `fmincon` (M), `fminimax` (M), `fminunc` (M), `fseminf` (M), `lsqcurvefit` (M), `lsqnonlin` (M)	`Directional derivative` See "Displaying Iterative Output" on page 2-79.
`firstorderopt`	First-order optimality (depends on algorithm). Final value equals optimization function output `output.firstorderopt`.	`fgoalattain` (M), `fmincon` (B), `fminimax` (M), `fminunc` (M), `fseminf` (M), `lsqcurvefit` (B), `lsqnonlin` (B)	`First-order optimality` See "Displaying Iterative Output" on page 2-79.

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`funccount`	Cumulative number of function evaluations. Final value equals optimization function output `output.funcCount`.	`fgoalattain` (M), `fminbnd` (B), `fmincon` (B), `fminimax` (M), `fminsearch` (B), `fminunc` (B), `fzero` (B), `fseminf` (M), `lsqcurvefit` (B), `lsqnonlin` (B)	`F-count` See "Displaying Iterative Output" on page 2-79.
`fval`	Function value at current point. Final value equals optimization function output `fval`.	`fgoalattain` (M), `fminbnd` (B), `fmincon` (B), `fminimax` (M), `fminsearch` (B), `fminunc` (B), `fseminf` (M), `fzero` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	`f(x)` See "Displaying Iterative Output" on page 2-79.
`gradient`	Current gradient of objective function — either analytic gradient if you provide it or finite-differencing approximation. Final value equals optimization function output `grad`.	`fgoalattain` (M), `fmincon` (B), `fminimax` (M), `fminunc` (M), `fseminf` (M), `lsqcurvefit` (B), `lsqnonlin` (B)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`iteration`	Iteration number — starts at `0`. Final value equals optimization function output `output.iterations`.	`fgoalattain` (M), `fminbnd` (B), `fmincon` (B), `fminimax` (M), `fminsearch` (B), `fminunc` (B), `fseminf` (M), `fzero` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	`Iteration` See "Displaying Iterative Output" on page 2-79.
`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. Final value equals optimization function output `lambda`.	`fgoalattain` (M), `fmincon` (M), `fminimax` (M), `fseminf` (M), `lsqcurvefit` (M), `lsqnonlin` (M)	None
`positivedefinite`	• `0` if algorithm detects negative curvature while computing Newton step • `1` otherwise	`fmincon` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
procedure	Procedure messages	fgoalattain (M), fminbnd (B), fmincon (M), fminimax (M), fminsearch (B), fseminf (M), fzero (B),	Procedure See "Displaying Iterative Output" on page 2-79.
ratio	Ratio of change in the objective function to change in the quadratic approximation	fmincon (L), lsqcurvefit (L), lsqnonlin (L)	None
residual	2-norm of the residual squared. Final value equals optimization function output residual.	lsqcurvefit (B), lsqnonlin (B)	Residual See "Displaying Iterative Output" on page 2-79.
searchdirection	Search direction	fgoalattain (M), fmincon (M), fminimax (M), fminunc (M), fseminf (M), lsqcurvefit (M), lsqnonlin (M)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`stepsize`	Current step size. Final value equals optimization function output `options.stepsize`.	`fgoalattain` (M), `fmincon` (B), `fminimax` (M), `fminunc` (B), `fseminf` (M), `lsqcurvefit` (B), `lsqnonlin` (B)	`Step-size` See "Displaying Iterative Output" on page 2-79.
`trustregionradius`	Radius of trust region	`fmincon` (L), `lsqcurvefit`, `lsqnonlin` (L)	`Trust-region radius` See "Displaying Iterative Output" on page 2-79.

Degeneracy. The value of the field `degenerate`, which measures the degeneracy of the current optimization point `x`, is defined as follows. First, define a vector `r`, of the same size as `x`, for which `r(i)` is the minimum distance from `x(i)` to the *i*th entries of the lower and upper bounds, `lb` and `ub`. That is,

```
r = min(abs(ub-x, x-lb))
```

Then the value of `degenerate` is the minimum entry of the vector `r + abs(grad)`, where `grad` is the gradient of the objective function. The value of `degenerate` is 0 if there is an index `i` for which both of the following are true:

- `grad(i) = 0`
- `x(i)` equals the *i*th entry of either the lower or upper bound.

States of the Algorithm

The following table lists the possible values for `state`:

State	Description
`'init'`	The algorithm is in the initial state before the first iteration.
`'interrupt'`	The algorithm is in some computationally expensive part of the iteration. In this state, the output function can interrupt the current iteration of the optimization. At this time, the values of `x` and `optimValues` are the same as at the last call to the output function in which `state=='iter'`.
`'iter'`	The algorithm is at the end of an iteration.
`'done'`	The algorithm is in the final state after the last iteration.

The following code illustrates how the output function might use the value of `state` to decide which tasks to perform at the current iteration.

```
switch state
    case 'iter'
          % Make updates to plot or guis as needed
    case 'interrupt'
          % Probably no action here. Check conditions to see
          % whether optimization should quit.
    case 'init'
          % Setup for plots or guis
    case 'done'
          % Cleanup of plots, guis, or final plot
otherwise
end
```

Stop Flag

The output argument `stop` is a flag that is `true` or `false`. The flag tells the optimization function whether the optimization should quit or continue. The following examples show typical ways to use the `stop` flag.

Stopping an Optimization Based on Data in optimValues. The output function can stop an optimization at any iteration based on the current data in `optimValues`. For example, the following code sets `stop` to `true` if the directional derivative is less than `.01`:

```
function stop = outfun(x, optimValues)
stop = false;
% Check if directional derivative is less than .01.
if optimValues.directionalderivative < .01
    stop = true;
end
```

Stopping an Optimization Based on GUI Input. If you design a GUI to perform optimizations, you can make the output function stop an optimization when a user presses a **Stop** button on the GUI. The following code shows how to do this, assuming that the **Stop** button callback stores the value `true` in the `optimstop` field of a `handles` structure called `hObject`.

```
function stop = outfun(x)
stop = false;
% Check if user has requested to stop the optimization.
stop = getappdata(hObject,'optimstop');
```

Functions — Alphabetical List

This section contains function reference pages listed alphabetically.

bintprog

Purpose

Solve binary integer programming problems of the form

$$\underset{x}{\text{minimize}} \quad f' \cdot x \qquad \text{such that} \qquad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \end{aligned}$$

where f, b, and beq are vectors, A and Aeq are matrices, and the solution x is required to be a binary integer vector — that is, its entries can only take on the values 0 or 1.

Syntax

```
x = bintprog(f)
x = bintprog(f, A, b)
x = bintprog(f, A, b, Aeq, beq)
x = bintprog(f, A, b, Aeq, beq, x0)
x = bintprog(f, A, b, Aeq, beq, x0, options)
[x, fval] = bintprog(...)
[x,fval, exitflag] = bintprog(...)
[x, fval, exitflag, output] = bintprog(...)
```

Description

`x = bintprog(f)` solves the binary integer programming problem

$$\underset{x}{\text{minimize}} \quad f' \cdot x$$

`x = bintprog(f, A, b)` solves the binary integer programming problem

$$\underset{x}{\text{minimize}} \quad f' \cdot x \qquad \text{such that} \qquad A \cdot x \le b$$

`x = bintprog(f, A, b, Aeq, beq)` solves the preceding problem with the additional equality constraint.

$$Aeq \cdot x = beq$$

`x = bintprog(f, A, b, Aeq, beq, x0)` sets the starting point for the algorithm to `x0`. If `x0` is not in the feasible region, `bintprog` uses the default initial point.

`x = bintprog(f, A, b, Aeq, Beq, x0, options)` minimizes with the default optimization options replaced by values in the structure `options`, which you can create using the function `optimset`.

[x, fval] = bintprog(...) returns fval, the value of the objective function at x.

[x,fval, exitflag] = bintprog(...) returns exitflag that describes the exit condition of bintprog. See "Output Arguments" on page 5-27.

[x, fval, exitflag, output] = bintprog(...) returns a structure output that contains information about the optimization. See "Output Arguments" on page 5-27.

Input Arguments

The following table lists the input arguments for bintprog. "Function Arguments" on page 5-5 contains general descriptions of input arguments for optimization functions.

f	Vector containing the coefficients of the linear objective function
A	Matrix containing the coefficients of the linear inequality constraints $A \cdot x \leq b$
b	Vector corresponding to the right-hand side of the linear inequality constraints
Aeq	Matrix containing the coefficients of the linear equality constraints $Aeq \cdot x = beq$
beq	Vector containing the constants of the linear equality constraints
x0	Initial point for the algorithm
options	Options structure containing options for the algorithm.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by bintprog. This section provides specific details for the arguments exitflag and output:

exitflag Integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated.

1	Function converged to a solution x.
0	Number of iterations exceeded options.MaxIter.

`-2`	The problem is infeasible.
`-4`	Number of searched nodes exceeded `options.MaxNodes`.
`-5`	Search time exceeded `options.MaxTime`.
`-6`	Number of iterations the LP-solver performed at a node to solve the LP-relaxation problem exceeded `options.MaxRLP`.

`output` Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken
`nodes`	Number of nodes searched
`time`	Execution time of the algorithm
`algorithm`	Algorithm used
`message`	Reason the algorithm terminated

Options

Optimization options used by `bintprog`. You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

`BranchStrategy` Strategy the algorithm uses to select the branch variable in the search tree — see "Branching" on page 5-30. The choices are

- `'mininfeas'` — Choose the variable with the minimum integer infeasibility, that is, the variable whose value is closest to `0` or `1` but not equal to `0` or `1`.
- `'maxinfeas'` — Choose the variable with the maximum integer infeasibility, that is, the variable whose value is closest to 0.5 (default).

The infeasibility

`Diagnostics` Display diagnostic information about the function

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`DispNodeInterval`	Node display interval
`MaxIter`	Maximum number of iterations allowed
`MaxNodes`	Maximum number of solutions, or nodes, the function searches
`MaxRLPIter`	Maximum number of iterations the LP-solver performs to solve the LP-relaxation problem at each node
`MaxTime`	Maximum amount of time in seconds the function runs
`NodeSearchStrategy`	Strategy the algorithm uses to select the next node to search in the search tree — see “Branching” on page 5-30. The choices are ▪ `'df'` — Depth first search strategy. At each node in the search tree, if there is child node one level down in the tree that has not already been explored, the algorithm chooses one such child to search. Otherwise, the algorithm moves to the node one level up in the tree and chooses a child node one level down from that node. ▪ `'bn'` — Best node search strategy, which chooses the node with lowest bound on the objective function.
`TolCon`	Termination tolerance on the constraint violation
`TolFun`	Termination tolerance on the function value
`TolXInteger`	Tolerance within which the value of a variable is considered to be integral
`TolRLPFun`	Termination tolerance on the function value of a linear programming relaxation problem

Algorithm

`bintprog` uses a linear programming (LP)-based branch-and-bound algorithm to solve binary integer programming problems. The algorithm searches for an optimal solution to the binary integer programming problem by solving a series

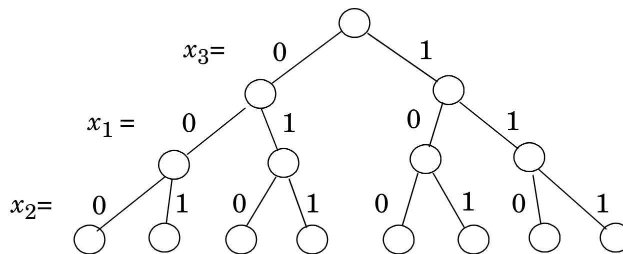
of *LP-relaxation* problems, in which the binary integer requirement on the variables is replaced by the weaker constraint $0 \le x \le 1$. The algorithm

- Searches for a binary integer feasible solution
- Updates the best binary integer feasible point found so far as the search tree grows
- Verifies that no better integer feasible solution is possible by solving a series of linear programming problems

The following sections describe the branch-and-bound method in greater detail.

Branching

The algorithm creates a search tree by repeatedly adding constraints to the problem, that is, "branching." At a branching step, the algorithm chooses a variable x_j whose current value is not an integer and adds the constraint $x_j = 0$ to form one branch and the constraint $x_j = 1$ to form the other branch. This process can be represented by a binary tree, in which the nodes represent the added constraints. The following picture illustrates a complete binary tree for a problem that has three variables, x_1, x_2, and x_3. Note that, in general, the order of the variables going down the levels in the tree is not the usual order of their subscripts



Deciding Whether to Branch

At each node, the algorithm solves an LP-relaxation problem using the constraints at that node and decides whether to branch or to move to another node depending on the outcome. There are three possibilities:

- If the LP-relaxation problem at the current node is infeasible or its optimal value is greater than that of the best integer point, the algorithm removes

the node from the tree, after which it does not search any branches below that node. The algorithm then moves to a new node according to the method you specify in `NodeSearchStrategy` option.

- If the algorithm finds a new feasible integer point with lower objective value than that of the best integer point, it updates the current best integer point and moves to the next node.
- If the LP-relaxation problem is optimal but not integer and the optimal objective value of the LP relaxation problem is less than the best integer point, the algorithm branches according to the method you specify in the `BranchStrategy` option.

See "Options" on page 5-28 for a description of the `NodeSearchStrategy` and `BranchStrategy` options.

Bounds

The solution to the LP-relaxation problem provides a lower bound for the binary integer programming problem. If the solution to the LP-relaxation problem is already a binary integer vector, it provides an upper bound for the binary integer programming problem.

As the search tree grows more nodes, the algorithm updates the lower and upper bounds on the objective function, using the bounds obtained in the bounding step. The bound on the objective value serves as the threshold to cut off unnecessary branches.

Limits for the Algorithm

The algorithm for bintprog could potentially search all 2^n binary integer vectors, where n is the number of variables. As a complete search might take a very long time, you can limit the search using the following options

- `MaxNodes` — Maximum number of nodes the algorithm searches
- `MaxRLPIter` — Maximum number of iterations the LP-solver performs at any node
- `MaxTime` — Maximum amount of time in seconds the algorithm runs

See "Options" on page 5-28 for more information.

Example

To minimize the function

$$f(x) = -9x_1 - 5x_2 - 6x_3 - 4x_4$$

subject to the constraints

$$\begin{bmatrix} 6 & 3 & 5 & 2 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \le \begin{bmatrix} 9 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

where x_1, x_2, x_3, and x_4 are binary integers, enter the following commands:

```
f = [-9; -5; -6; -4];
A = [6 3 5 2; 0 0 1 1; -1 0 1 0; 0 -1 0 1];
b = [9; 1; 0; 0];
x = bintprog(f,A,b)
Optimization terminated successfully.

x =

     1
     1
     0
     0
```

See Also

`optimset`

References

[1] Wolsey, Laurence A., *Integer Programming*, John Wiley & Sons, 1998.

[2] Nemhauser, George L. and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.

[3] Hillier, Frederick S. and Lieberman Gerald J., Introduction to Operations Research, McGraw-Hill, 2001.

Purpose

Solve multiobjective goal attainment problem

$$\underset{x,\gamma}{\text{minimize}}\ \gamma \quad \text{such that} \quad \begin{aligned} &F(x) - weight \cdot \gamma \le goal \\ &c(x) \le 0 \\ &ceq(x) = 0 \\ &A \cdot x \le b \\ &Aeq \cdot x = beq \\ &lb \le x \le ub \end{aligned}$$

where x, $weight$, $goal$, b, beq, lb, and ub are vectors, A and Aeq are matrices, and $c(x)$, $ceq(x)$, 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,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 in `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 options specified in the structure `options`. Use `optimset` to set these options.

`[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`.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fgoalattain`. This section provides function-specific details for `fun`, `goal`, `nonlcon`, `options`, and `weight`:

`fun` The function to be minimized. `fun` is a function that accepts a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fgoalattain(@myfun,x0,goal,weight)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x)
F = ...          % Compute function values at x.
```

`fun` can also be a function handle for an anonymous function.

```
x = fgoalattain(@(x)sin(x.*x),x0,goal,weight);
```

To make an objective function as near as possible to a goal value, (i.e., neither greater than nor less than) use `optimset` to set the `GoalsExactAchieve` option 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* the `GradObj` option 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 `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 values at x
if nargout > 1      % Two output arguments
   G = ...          % Gradients evaluated at x
end
```

The gradient consists of the partial derivative dF/dx of each F at the point x. If F is a vector of length m and x has length n, where n is the length of x0, 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 jth column of G is the gradient of the jth 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 the nonlinear equality constraints ceq(x) = 0. The function nonlcon accepts a vector x and returns two vectors c and ceq. The vector c contains the nonlinear inequalities evaluated at x, and ceq contains the nonlinear equalities evaluated at x. The function nonlcon can be specified as a function handle.

```
x = fgoalattain(@myfun,x0,goal,weight,A,b,Aeq,beq,...
                lb,ub,@mycon)
```

where mycon is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...         % compute nonlinear inequalities at x.
ceq = ...       % compute nonlinear equalities at x.
```

If the gradients of the constraints can also be computed *and* the GradConstr option 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`, where `n` is the length of `x0`, 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)`).

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

`options` "Options" on page 5-39 provides the function-specific details for the `options` values.

`weight` A weighting vector to control the relative under-attainment or overattainment of the objectives in `fgoalattain`. When the values of `goal` are *all nonzero*, to ensure the same percentage of under- or overattainment 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 is zero causes 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 option and put that objective as the first element of the vector returned by fun (see the preceding description of fun and options).

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by fgoalattain. This section provides function-specific details for attainfactor, exitflag, lambda, and output:

attainfactor — The amount of over- or underachievement of the goals. If attainfactor is negative, the goals have been overachieved; if attainfactor is positive, the goals have been underachieved.

exitflag — Integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated.

exitflag	Reason
1	Function converged to a solutions x.
4	Magnitude of the search direction less than the specified tolerance and constraint violation less than options.TolCon
5	Magnitude of directional derivative less than the specified tolerance and constraint violation less than options.TolCon
0	Number of iterations exceeded options.MaxIter or number of function evaluations exceeded options.FunEvals
-1	Algorithm was terminated by the output function.
-2	No feasible point was found.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are	
	`lower`	Lower bounds `lb`
	`upper`	Upper bounds `ub`
	`ineqlin`	Linear inequalities
	`eqlin`	Linear equalities
	`ineqnonlin`	Nonlinear inequalities
	`eqnonlin`	Nonlinear equalities
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`algorithm`	Algorithm used

Options

Optimization options used by `fgoalattain`. You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

`DerivativeCheck`	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives.
`Diagnostics`	Display 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'` (default) displays just the final output.

GoalsExactAchieve	Specifies the number of objectives for which it is required for the objective fun to equal the goal goal. Such objectives should be partitioned into the first few elements of F.
GradConstr	Gradient for the constraints defined by the user. See the preceding description of nonlcon to see how to define the gradient in nonlcon.
GradObj	Gradient for the objective function defined by user. See the preceding description of fun to see how to define the gradient in fun. You must provide the gradient 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'.
OutputFcn	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
TolCon	Termination tolerance on the constraint violation.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.

Examples

Consider a linear system of differential equations.

An output feedback controller, K, is designed producing a closed loop system

$$\dot{x} = (A + BKC)x + Bu$$
$$y = Cx$$

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 is initialized as

```
goal = [-5,-3,-1];
```

To ensure the same percentage of under- or overattainment 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(@(K)eigfun(K,A,B,C)...
    goal,weight,[],[],[],[],lb,ub,[],options)
```

You can run this example 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 overachieved 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 overattainment is met exactly. Three of the lower bound constraints are also active.

In the preceding 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, use `optimset` to set the `GoalsExactAchieve` option to the number of objectives for which this is required.

Consider the preceding 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 the `GoalsExactAchieve` option set to `3`.

```
options = optimset('GoalsExactAchieve',3);
[K,fval,attainfactor] = fgoalattain(...
   @(K)eigfun(K,A,B,C),K0,goal,weight,[],[],[],[],lb,ub,[],...
options)
```

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 because 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 [3]. This entails the construction of a set of *goal* values for the objective functions. Multiobjective optimization is discussed fully in the "Standard Algorithms" chapter.

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 not known whether the objectives will reach the goals (under attainment) or be minimized less than the goals (overattainment). A weighting vector, *weight*, controls the relative underattainment or overattainment of the objectives.

`fgoalattain` uses a sequential quadratic programming (SQP) method, which is described fully in the "Standard Algorithms" chapter. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [1] and [4]) is used together with the merit function proposed by [5], [6]. The line search is terminated when either merit function shows improvement. A modified Hessian, which takes advantage of the special structure of the problem, is also used (see [1] and [[4]). A full description of the modifications used is found in "Goal Attainment Method" on page 3-47 in "Introduction to Algorithms." Setting the `MeritFunction` option to `'singleobj'` with

```
options = optimset(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 overattainment in the goals.

See also "SQP Implementation" on page 3-30 for more details on the algorithm used and the types of procedures displayed under the `Procedures` heading when the `Display` option is set to `'iter'`.

Limitations

The objectives must be continuous. `fgoalattain` might give only local solutions.

See Also

`@(function_handle)`, `fmincon`, `fminimax`, `optimset`

References

[1] 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.

[2] 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.

[3] Gembicki, F.W., "Vector Optimization for Control with Performance and Parameter Sensitivity Indices," Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, OH, 1974.

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

[5] Han, S.P., "A Globally Convergent Method For Nonlinear Programming," *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

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

Purpose

Find a minimum of a function of one variable on a fixed interval

$$\min_{x} f(x) \quad \text{such that} \quad x_1 \le x \le x_2$$

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,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` attempts to find a 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 options specified in the structure `options`. Use `optimset` to set these options.

`[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.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fminbnd`. This section provides function-specific details for `fun` and `options`:

`fun`	The function to be minimized. `fun` is a function that accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminbnd(@myfun,x1,x2)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...         % Compute function value at x.
```

`fun` can also be a function handle for an anonymous function.

```
x = fminbnd(@(x)sin(x*x),x1,x2);
```

`options`	"Options" on page 5-47 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fminbnd`. This section provides function-specific details for `exitflag` and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	Function converged to a solution `x`.
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`
-1	Algorithm was terminated by the output function.
-2	The bounds are inconsistent.

`output` Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations
`algorithm`	Algorithm used

`message`	Exit message

Options

Optimization options used by `fminbnd`. You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no warning.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify a user-defined function that the optimization function calls at each iteration. See "Calling an Output Function Iteratively" on page 2-85.
`TolX`	Termination tolerance on `x`.

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
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbind`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

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` might 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

`@` (`function_handle`), `fminsearch`, `fmincon`, `fminunc`, `optimset`, anonymous functions

References

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

fmincon

Purpose

Find a minimum of a constrained nonlinear multivariable function

$$\min_{x} f(x) \text{ subject to}$$

$$c(x) \le 0$$
$$ceq(x) = 0$$
$$A \cdot x \le b$$
$$Aeq \cdot x = beq$$
$$lb \le x \le ub$$

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,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` attempts to find a 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 attempts to find 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 in `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 options specified in the structure `options`. Use `optimset` to set these options. Set `nonlcon = []` if there are no nonlinear inequality or equality constraints.

`[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 at the solution `x`. See "Hessian" on page 5-57

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fmincon`. This "Arguments" section provides function-specific details for `fun`, `nonlcon`, and `options`:

`fun` The function to be minimized. `fun` is a function that accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fmincon(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fmincon(@(x)norm(x)^2,x0,A,b);
```

If the gradient of `fun` can also be computed *and* the `GradObj` option 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 consists of 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`.

If the Hessian matrix can also be computed *and* the `Hessian` option 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` you 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
end
```

The Hessian matrix is the second partial derivatives matrix of f at the point x. That is, the (i,j)th 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 the nonlinear equality constraints `ceq(x) = 0`. The function `nonlcon` accepts a vector `x` and returns two vectors `c` and `ceq`. The vector `c` contains the nonlinear inequalities evaluated at `x`, and `ceq` contains the nonlinear equalities evaluated at `x`. The function `nonlcon` can be specified as a function handle.

```
x = fmincon(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)
```

where `mycon` is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x.
ceq = ...   % Compute nonlinear equalities at x.
```

If the gradients of the constraints can also be computed *and* the `GradConstr` option 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`).

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

```
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`, where `n` is the length of `x0`, 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` "Options" on page 5-57 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fmincon`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	First order optimality conditions were satisfied to the specified tolerance.
2	Change in `x` was less than the specified tolerance.
3	Change in the objective function value was less than the specified tolerance.
4	Magnitude of the search direction was less than the specified tolerance and constraint violation was less than `options.TolCon`.

	5	Magnitude of directional derivative was less than the specified tolerance and constraint violation was less than options.TolCon.
	0	Number of iterations exceeded options.MaxIter or number of function evaluations exceeded options.FunEvals
	-1	Algorithm was terminated by the output function.
	-2	No feasible point was found.
grad	Gradient at x	
hessian	Hessian at x	
lambda	Structure containing the Lagrange multipliers at the solution x (separated by constraint type). The fields of the structure are	
	lower	Lower bounds lb
	upper	Upper bounds ub
	ineqlin	Linear inequalities
	eqlin	Linear equalities
	ineqnonlin	Nonlinear inequalities
	eqnonlin	Nonlinear equalities
output	Structure containing information about the optimization. The fields of the structure are	
	iterations	Number of iterations taken
	funcCount	Number of function evaluations
	algorithm	Algorithm used.
	cgiterations	Number of PCG iterations (large-scale algorithm only)
	stepsize	Final step size taken (medium-scale algorithm only)

`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient. For large-scale problems with only linear equalities, the first-order optimality is the infinity norm of the *projected* gradient (i.e. the gradient projected onto the nullspace of `Aeq`).

Hessian

`fmincon` computes the output argument `hessian` as follows:

- When using the medium-scale algorithm, the function computes a quasi-Newton approximation to the Hessian of the Lagrangian at `x`.
- When using the large-scale algorithm, the function uses
  - `options.Hessian`, if you supply it, to compute the Hessian at `x`
  - A finite-difference approximation to the Hessian at `x`, if you supply only the gradient. Note that because the large-scale algorithm does not take nonlinear constraints, the Hessian of the Lagrangian is the same as the Hessian of the objective function.

Options

Optimization options used by `fmincon`. Some options apply to all algorithms, some are only relevant when you are using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm.You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

The `LargeScale` option specifies a preference for which algorithm to use. It is only a preference because certain conditions must be met to use the large-scale algorithm. For `fmincon`, you must provide the gradient (see the preceding description of `fun` to see how) or else the medium-scale algorithm is used:

`LargeScale`	Use the large-scale algorithm if possible when set to `'on'`. Use the medium-scale algorithm when set to `'off'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`DerivativeCheck`	Compare user-supplied derivatives (gradients of the objective and constraints) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`GradObj`	Gradient for the objective function defined by the user. See the preceding description of `fun` to see how to define the gradient in `fun`. You must provide the gradient 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
`OutputFcn`	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
`TolFun`	Termination tolerance on the function value.
`TolCon`	Termination tolerance on the constraint violation.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

`Hessian`	If `'on'`, `fmincon` uses a user-defined Hessian (defined in `fun`), or Hessian information (when using `HessMult`), for the objective function. If `'off'`, `fmincon` approximates the Hessian using finite differences.
`HessMult`	Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product `H*Y` without actually forming `H`. The function is of the form

```
W = hmfun(Hinfo,Y,p1,p2,...)
```

where `Hinfo` and possibly the additional parameters `p1,p2,...` contain the matrices used to compute `H*Y`.

The first argument must be the same as the third argument returned by the objective function `fun`, for example by

```
[f,g,Hinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `W = H*Y` although `H` is not formed explicitly. `fminunc` uses `Hinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `hmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for the parameters.

Note `'Hessian'` must be set to `'on'` for `Hinfo` to be passed from `fun` to `hmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for an example.

`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 is 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 following).
`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.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

`DiffMaxChange`	Maximum change in variables for finite-difference gradients.
`DiffMinChange`	Minimum change in variables for finite-difference gradients.
`MaxSQPIter`	Maximum number of SQP iterations allowed

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 less than or equal to 0.

```
A*x-b=
    -72
      0
```

Notes

Large-Scale Optimization. To use the large-scale method, you must

- Supply the gradient in `fun`
- Set `GradObj` to `'on'` in `options`
- Specify the feasible region using one, but not both, of the following types of constraints:
  - Upper and lower bounds constraints
  - Linear equality constraints, in which the equality constraint matrix `Aeq` cannot have more rows than columns. `Aeq` is typically sparse.

You cannot use inequality constraints with the large-scale algorithm. If the preceding conditions are not met, `quadprog` reverts to the medium-scale algorithm.

The function `fmincon` returns a warning if no gradient is provided and the `LargeScale` option is not `'off'`. `fmincon` permits $g(x)$ to be an approximate gradient but this option is not recommended; the numerical behavior of most optimization methods is considerably more robust when the true gradient is used. See Table 2-4, Large-Scale Problem Coverage and Requirements, on page 2-42, for more information on what problem formulations are covered and what information you must be provide.

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` option in `options`), `fmincon` computes 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 solution with matrix

  $$\overline{Aeq} = Aeq^T R^{-T}$$

  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 $\overline{Aeq}$.

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 displayed under the `Procedures` heading (when you ask for output by setting the `Display` option to `'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 displayed under the `Procedures` heading.

Algorithm

Large-Scale Optimization. The large-scale algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1], [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. `fmincon` uses a sequential quadratic programming (SQP) method. In this method, the function solves a quadratic programming (QP) subproblem at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula (see `fminunc`, references [7], [8]).

A line search is performed using a merit function similar to that proposed by [4], [5], and [6]. The QP subproblem is solved using an active set strategy similar to that described in [3]. A full description of this algorithm is found in "Constrained Optimization" on page 3-27 in "Introduction to Algorithms."

See also "SQP Implementation" on page 3-30 in "Introduction to Algorithms" for more details on the algorithm used.

Limitations

`fmincon` only handles real variables.

The function to be minimized and the constraints must both be continuous. `fmincon` might 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.

The large-scale method does 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.

See Also

`@(function_handle)`, `fminbnd`, `fminsearch`, `fminunc`, `optimset`

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

Purpose

Solve the minimax problem

$$\min_{x} \max_{\{F_i\}} \{F_i(x)\} \quad \text{such that} \quad \begin{aligned} c(x) &\le 0 \\ ceq(x) &= 0 \\ A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

where x, b, beq, lb, and ub are vectors, A and Aeq are matrices, and $c(x)$, $ceq(x)$, 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,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 might 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 in `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 options specified in the structure `options`. Use `optimset` to set these options.

`[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`.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fminimax`. This section provides function-specific details for `fun`, `nonlcon`, and `options`:

`fun` The function to be minimized. `fun` is a function that accepts a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminimax(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x)
F = ...            % Compute function values at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fminimax(@(x)sin(x.*x),x0);
```

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 use `optimset` to set the `MinAbsMax` option to be the number of such objectives.

If the gradient of the objective function can also be computed *and* the `GradObj` option 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 consists of the partial derivative dF/dx of each F at the point x. If F is a vector of length m and x has length n, where n is the length of x0, 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 jth column of G is the gradient of the jth objective function F(j)).

nonlcon The function that computes the nonlinear inequality constraints c(x) <= 0 and nonlinear equality constraints ceq(x) = 0. The function nonlcon accepts a vector x and returns two vectors c and ceq. The vector c contains the nonlinear inequalities evaluated at x, and ceq contains the nonlinear equalities evaluated at x. The function nonlcon can be specified as a function handle.

```
x = fminimax(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)
```

where mycon is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x
ceq = ...   % Compute nonlinear equalities at x
```

If the gradients of the constraints can also be computed *and* the GradConstr option 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`, where `n` is the length of `x0`, 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)`).

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

`options` — "Options" on page 5-70 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fminimax`. This section provides function-specific details for `exitflag`, `lambda`, `maxfval`, and `output`:

`exitflag` — Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

Value	Reason
1	Function converged to a solution `x`.
4	Magnitude of the search direction less than the specified tolerance and constraint violation less than `options.TolCon`
5	Magnitude of directional derivative less than the specified tolerance and constraint violation less than `options.TolCon`
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
-1	Algorithm was terminated by the output function.
-2	No feasible point was found.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are

`lower`	Lower bounds `lb`
`upper`	Upper bounds `ub`
`ineqlin`	Linear inequalities
`eqlin`	Linear equalities
`ineqnonlin`	Nonlinear inequalities
`eqnonlin`	Nonlinear equalities

`maxfval`	Maximum of the function values evaluated at the solution `x`, that is, `maxfval = max{fun(x)}`.
`output`	Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken.
`funcCount`	Number of function evaluations.
`algorithm`	Algorithm used.

Options

Optimization options used by `fminimax`. You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

`DerivativeCheck`	Compare user-supplied derivatives (gradients of the objective or constraints) to finite-differencing derivatives.
`Diagnostics`	Display 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'` (default) displays just the final output.
`GradConstr`	Gradient for the constraints defined by user. See the preceding description of `nonlcon` to see how to define the gradient in `nonlcon`.
`GradObj`	Gradient for the objective function defined by user. See the preceding description of `fun` to see how to define the gradient in `fun`. You must provide the gradient 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 the goal attainment/minimax merit function if set to `'multiobj'`. Use the `fmincon` merit function if set to `'singleobj'`.
`MinAbsMax`	Number of *F(x)* to minimize the worst case absolute values.
`OutputFcn`	Specify a user-defined function that is called after each iteration of an optimization (medium scale algorithm only). See "Output Function" on page 5-15.
`TolCon`	Termination tolerance on the constraint violation.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.

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

You can set the number of objectives for which the worst case absolute values of `F` are minimized in the `MinAbsMax` option using `optimset`. You should partition these objectives into the first elements of `F`.

For example, consider the preceding problem, which requires finding values of `x` that minimize the maximum absolute value of

$$\left[\; f_1(x)\;,\; f_2(x)\;,\; f_3(x)\;,\; f_4(x)\;,\; f_5(x)\;\right]$$

Solve this problem 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(@myfun,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 displayed under the `Procedures` heading (when the `Display` option is set to `'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 displayed under the `Procedures` heading.

Algorithm

`fminimax` uses a sequential quadratic programming (SQP) method [1]. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [2] and [4]) is used together with the merit function proposed by [3] and [5]. The line search is terminated when either merit function shows improvement. The function uses a modified Hessian that takes advantage of the special structure of this problem. Using `optimset` to set the `MeritFunction` option to`'singleobj'` uses the merit function and Hessian used in `fmincon`.

See also "SQP Implementation" on page 3-30 for more details on the algorithm used and the types of procedures printed under the `Procedures` heading when you set the `Display` option to`'iter'`.

Limitations

The function to be minimized must be continuous. `fminimax` might only give local solutions.

See Also

`@` (`function_handle`), `fgoalattain`, `lsqnonlin`, `optimset`

References

[1] 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.

[2] Grace, A.C.W., "Computer-Aided Control System Design Using Optimization Techniques," Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

[3] Han, S.P., "A Globally Convergent Method For Nonlinear Programming," *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.

[4] Madsen, K. and H. Schjaer-Jacobsen, "Algorithms for Worst Case Tolerance Optimization," *IEEE Trans. of Circuits and Systems*, Vol. CAS-26, Sept. 1979.

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

Purpose

Find a 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,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` attempts to find a 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 attempts to find 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 options specified in the structure `options`. Use `optimset` to set these 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.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fminsearch`. This section provides function-specific details for `fun` and `options`:

`fun`	The function to be minimized. `fun` is a function that accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminsearch(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fminsearch(@(x)norm(x)^2,x0,A,b);
```

`options`	"Options" on page 5-77 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fminsearch`. This section provides function-specific details for `exitflag` and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	The function converged to a solution `x`.
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
-1	The algorithm was terminated by the output function.

`output` Structure containing information about the optimization. The fields of the structure are

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Options

Optimization options used by `fminsearch`. You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no warning.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.

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 anonymous function:

```
f = @(x)sin(x)+3;
x = fminsearch(f,2);
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1 Assign the value to `a`.

```
a = 1.5; % define parameter first
```

2 Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fminbnd(@(x) myfun(x,a),0,1)
```

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` might be more robust.

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. `fminsearch` might 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

`@`(`function_handle`), `fminbnd`, `fminunc`, `optimset`, anonymous functions

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9, Number 1, pp.112-147, 1998.

fminunc

Purpose

Find a 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,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` attempts to find a 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 attempts to find 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 options specified in the structure `options`. Use `optimset` to set these 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`. See "Hessian" on page 5-83.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fminunc`. This section provides function-specific details for `fun` and `options`:

`fun` The function to be minimized. `fun` is a function that accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminunc(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fminunc(@(x)norm(x)^2,x0);
```

If the gradient of `fun` can also be computed *and* the `GradObj` option 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* the Hessian option 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 you 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
end
```

The Hessian matrix is the second partial derivatives matrix of f at the point x. That is, the (i,j)th 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 "Options" on page 5-84 provides the function-specific details for the options values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by fminunc. This section provides function-specific details for exitflag and output:

exitflag Integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated.

1	Function converged to a solution x.
2	Change in x was smaller than the specified tolerance.

	3	Change in the objective function value was less than the specified tolerance.
	0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
	-1	Algorithm was terminated by the output function.
	-2	Line search could not sufficiently decrease the objective function along the current search direction.
`grad`	Gradient at `x`	
`hessian`	Hessian at `x`	
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`algorithm`	Algorithm used.
	`cgiterations`	Number of PCG iterations (large-scale algorithm only)
	`stepsize`	Final step size taken (medium-scale algorithm only)

Hessian

`fminunc` computes the output argument `hessian` as follows:

- When using the medium-scale algorithm, the function computes a finite-difference approximation to the Hessian at `x` using
  - The gradient `grad` if you supply it
  - The objective function `fun` if you do not supply the gradient
- When using the large-scale algorithm, the function uses
  - `options.Hessian`, if you supply it, to compute the Hessian at `x`

- A finite-difference approximation to the Hessian at x, if you supply only the gradient

Options

fminunc uses these optimization options. Some options apply to all algorithms, some are only relevant when you are using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm.You can use optimset to set or change the values of these fields in the options structure options. See "Optimization Options" on page 5-9, for detailed information.

The LargeScale option specifies a *preference* for which algorithm to use. It is only a preference, because certain conditions must be met to use the large-scale algorithm. For fminunc, you must provide the gradient (see the preceding description of fun) or else use the medium-scale algorithm.:

LargeScale	Use large-scale algorithm if possible when set to 'on'. Use medium-scale algorithm when set to 'off'.

Large-Scale and Medium-Scale Algorithms. These options are used by both the large-scale and medium-scale algorithms:

DerivativeCheck	Compare user-supplied derivatives (gradient) to finite-differencing derivatives.
Diagnostics	Display diagnostic information about the function to be minimized.
Display	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' (default) displays just the final output.
GradObj	Gradient for the objective function that you define. See the preceding description of fun to see how to define the gradient in fun. You must provide the gradient 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.

`OutputFcn`	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

`Hessian` — If `'on'`, `fminunc` uses a user-defined Hessian (defined in `fun`), or Hessian information (when using `HessMult`), for the objective function. If `'off'`, `fminunc` approximates the Hessian using finite differences.

`HessMult` — Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product `H*Y` without actually forming `H`. The function is of the form

```
W = hmfun(Hinfo,Y,p1,p2,...)
```

where `Hinfo` and possibly the additional parameters `p1,p2,...` contain the matrices used to compute `H*Y`.

The first argument must be the same as the third argument returned by the objective function `fun`, for example by

```
[f,g,Hinfo] = fun(x)
```

Y is a matrix that has the same number of rows as there are dimensions in the problem. W = H*Y although H is not formed explicitly. fminunc uses Hinfo to compute the preconditioner. The optional parameters p1, p2, ... can be any additional parameters needed by hmfun. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for the parameters.

Note 'Hessian' must be set to 'on' for Hinfo to be passed from fun to hmfun.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for an example.

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 is 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 "Algorithms" on page 5-89).
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.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

DiffMaxChange	Maximum change in variables for finite-difference gradients.
DiffMinChange	Minimum change in variables for finite-difference gradients.
HessUpdate	Method for choosing the search direction in the Quasi-Newton algorithm. The choices are • 'bfgs' • 'dfp' • 'steepdesc' See "Hessian Update" on page 3-10 for a description of these methods.
InitialHessMatrix	Initial quasi-Newton matrix. This option is only available if you set InitialHessType to 'user-supplied'. In that case, you can set InitialHessMatrix to one of the following: • scalar — the initial matrix is the scalar times the identity • vector — the initial matrix is a diagonal matrix with the entries of the vector on the diagonal.
InitialHessType	Initial quasi-Newton matrix type. The options are • {'identity'} • 'scaled-identity' • 'user-supplied'

Examples

Minimize the function $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2$.

To use an M-file, 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 *
   -0.7512    0.2479
fval =
  1.3818e-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 that the gradient value is available by creating an optimization options structure with the `GradObj` option 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 *
   0.1110   -0.8882
fval2 =
  6.2862e-031
```

To minimize the function `f(x) = sin(x) + 3` using an anonymous function

```
f = @(x)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, you must provide the gradient in `fun` (and set the `GradObj` option to `'on'` using `optimset`). A warning is given if no gradient is provided and the `LargeScale` option is not `'off'`.

Algorithms

Large-Scale Optimization. By default `fminunc` chooses the large-scale algorithm if the user supplies the gradient in `fun` (and the `GradObj` option is set to `'on'` using `optimset`). This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [2],[3]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See “Trust-Region Methods for Nonlinear Minimization” on page 4-2 and “Preconditioned Conjugate Gradients” on page 4-5.

Medium-Scale Optimization. `fminunc`, with the `LargeScale` option set to `off` with `optimset`, uses the BFGS Quasi-Newton method with a mixed quadratic and cubic line search procedure. This quasi-Newton method uses the BFGS ([1],[5],[8],[9]) formula for updating the approximation of the Hessian matrix. You can select the DFP ([4],[6],[7]) formula, which approximates the inverse Hessian matrix, by setting the `HessUpdate` option to `'dfp'` (and the `LargeScale` option to `'off'`). You can select a steepest descent method by setting `HessUpdate` to `'steepdesc'` (and `LargeScale` to `'off'`), although this is not recommended.

The default line search algorithm, i.e., when the `LineSearchType` option is set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. You can select a safeguarded cubic polynomial method by setting the `LineSearchType` option 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. "Standard Algorithms" on page 3-1 provides a full description of the algorithms.

Limitations

The function to be minimized must be continuous. `fminunc` might 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 2-4, Large-Scale Problem Coverage and Requirements, on page 2-42, for more information on what problem formulations are covered and what information must be provided.

See Also

`@(function_handle)`, `fminsearch`, `optimset`, anonymous functions

References

[1] Broyden, C.G., "The Convergence of a Class of Double-Rank Minimization Algorithms," *Journal Inst. Math. Applic*., Vol. 6, pp. 76-90, 1970.

[2] 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.

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

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

[6] Fletcher, R., "Practical Methods of Optimization," Vol. 1, *Unconstrained Optimization*, John Wiley and Sons, 1980.

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

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

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

fseminf

Purpose

Find a 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,fval] = fseminf(...)
[x,fval,exitflag] = fseminf(...)
[x,fval,exitflag,output] = fseminf(...)
[x,fval,exitflag,output,lambda] = fseminf(...)
```

Description

`fseminf` finds a 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 $w_i \in \Re^2$). Because 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 in `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 options specified in the structure `options`. Use `optimset` to set these options.

`[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`.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fseminf`. This section provides function-specific details for `fun`, `ntheta`, `options`, and `seminfcon`:

`fun` The function to be minimized. `fun` is a function that accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fseminf(@myfun,x0,ntheta,seminfcon)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function.

```
fun = @(x)sin(x''*x);
```

If the gradient of `fun` can also be computed *and* the `GradObj` option 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`	"Options" on page 5-98 provides the function-specific details for the `options` values.

`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`. The function `seminfcon` can be specified as a function handle.

```
x = fseminf(@myfun,x0,ntheta,@myinfcon)
```

where `myinfcon` is a MATLAB function such as

```
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 might or might 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 `i`th 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" on page 5-101). On the first iteration `S` is `NaN`, so that some initial sampling interval must be determined by `seminfcon`.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize `seminfcon`, if necessary.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fseminf`. This section provides function-specific details for `exitflag` , `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	Function converged to a solution `x`.
4	Magnitude of the search direction was less than the specified tolerance and constraint violation was less than `options.TolCon`.
5	Magnitude of directional derivative was less than the specified tolerance and constraint violation was less than `options.TolCon`.
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
-1	Algorithm was terminated by the output function.
-2	No feasible point was found.

`lambda` Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are

`lower`	Lower bounds `lb`
`upper`	Upper bounds `ub`
`ineqlin`	Linear inequalities
`eqlin`	Linear equalities
`ineqnonlin`	Nonlinear inequalities
`eqnonlin`	Nonlinear equalities

`output` Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations

algorithm	Algorithm used
stepsize	Final step size taken

Options

Optimization options used by fseminf. You can use optimset to set or change the values of these fields in the options structure options. See "Optimization Options" on page 5-9, for detailed information.

DerivativeCheck	Compare user-supplied derivatives (gradients) to finite-differencing derivatives.
Diagnostics	Display 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' (default) displays just the final output.
GradObj	Gradient for the objective function defined by the user. See the description of fun above to see how to define the gradient in fun. You must provide the gradient 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.
OutputFcn	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
TolCon	Termination tolerance on the constraint violation.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.

Notes

The optimization routine `fseminf` might vary the recommended sampling interval, `S`, set in `seminfcon`, during the computation because values other than the recommended interval might 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. Because the constraints must be in the form $K_i(x, w_i) \le 0$ you 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-0.5).^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 finite nonlinear 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
[x,fval] = fseminf(@myfun,x0,2,@mycon)
```

After eight iterations, the solution is

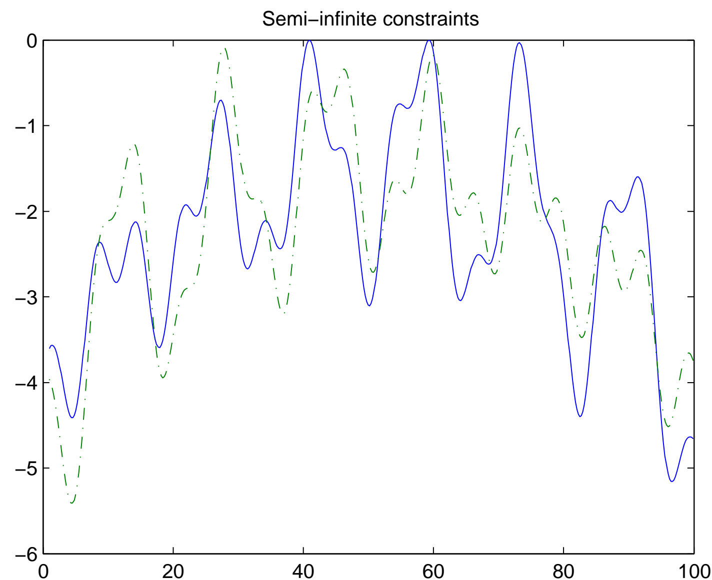
```
x =
     0.6673
     0.3013
     0.4023
```

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);  % Use initial sampling interval
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 constraints are on the constraint boundary.

The plot command inside 'mycon.m' slows down the computation. Remove this line to improve the speed.

Two-Dimensional Example

Find values of x that minimize

$$f(x) = (x_1 - 0.2)^2 + (x_2 - 0.2)^2 + (x_3 - 0.2)^2$$

where

$$K_1(x, w) = \sin(w_1 x_1)\cos(w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \ldots$$
$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) + -x_3 \le 1.5$$

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 $x = [0.25, 0.25, 0.25]$.

Note that the semi-infinite constraint is two-dimensional, that is, a matrix.

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 for the constraints, called `mycon.m`. Include code to draw the surface plot of the semi-infinite constraint each time `mycon` is called. This enables you to see how the constraint changes as `X` is being minimized.

```
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(wx*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
m = surf(wx,wy,K1,'edgecolor','none','facecolor','interp');
camlight headlight
title('Semi-infinite constraint')
drawnow
```

Next, invoke an optimization routine.

```
x0 = [0.25, 0.25, 0.25];    % Starting guess
[x,fval] = fseminf(@myfun,x0,1,@mycon)
```

After nine iterations, the solution is

```
x =
     0.2926    0.1874    0.2202
```

and the function value at the solution is

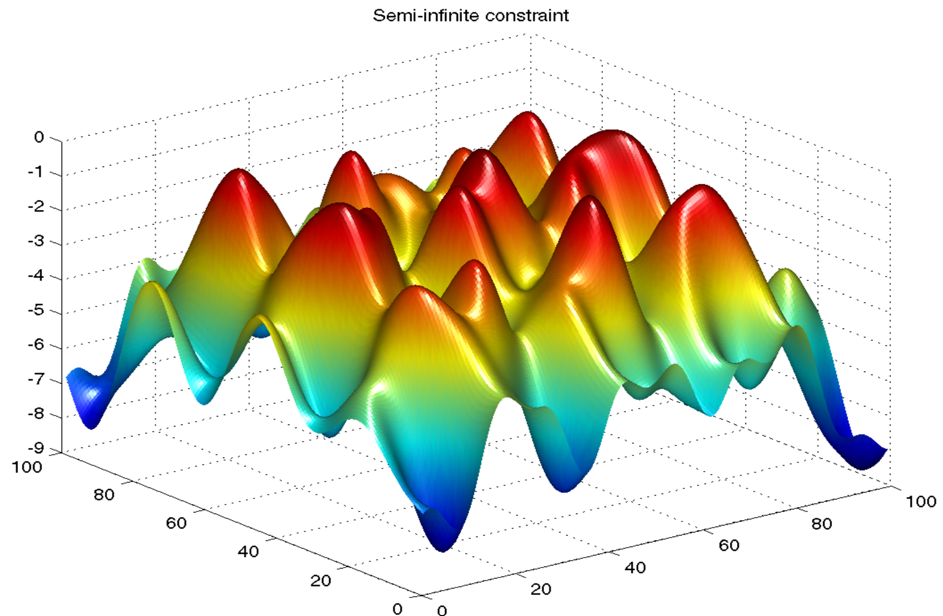
```
fval =
     0.0091
```

The goal was to minimize the objective $f(x)$ such that the semi-infinite constraint satisfied $K_1(x, w) \le 1.5$. Evaluating mycon at the solution x and looking at the maximum element of the matrix K1 shows the constraint is easily satisfied.

```
[c,ceq,K1] = mycon(x,[0.5,0.5]);  % Sampling interval 0.5
max(max(K1))

ans =
    -0.0027
```

This call to mycon produces the following surf plot, which shows the semi-infinite constraint at x.



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 the function needs to take more or fewer points. The function also evaluates the effectiveness of the interpolation 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 “SQP Implementation” on page 3-30 for more details on the algorithm used and the types of procedures displayed under the `Procedures` heading when the `Display` option is set to `'iter'` with `optimset`.

Limitations

The function to be minimized, the constraints, and semi-infinite constraints, must be continuous functions of `x` and `w`. `fseminf` might only give local solutions.

When the problem is not feasible, `fseminf` attempts to minimize the maximum constraint value.

See Also

`@` (`function_handle`), `fmincon`, `optimset`

fsolve

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,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 options specified in the structure `options`. Use `optimset` to set these 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`

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fsolve`. This section provides function-specific details for `fun` and `options`:

`fun` The nonlinear system of equations to solve. `fun` is a function that accepts a vector `x` and returns a vector `F`, the nonlinear equations evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fsolve(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x)
F = ...              % Compute function values at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fsolve(@(x)sin(x.*x),x0);
```

If the Jacobian can also be computed *and* the `Jacobian` option 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`, where `n` is the length of `x0`, 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` “Options” on page 5-109 provides the function-specific details for the `options` values.

Output Arguments

“Function Arguments” on page 5-5 contains general descriptions of arguments returned by `fsolve`. This section provides function-specific details for `exitflag` and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

`1`	Function converged to a solution `x`.
`2`	Change in `x` was smaller than the specified tolerance.
`3`	Change in the residual was smaller than the specified tolerance.
`4`	Magnitude of search direction was smaller than the specified tolerance.
`0`	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
`-1`	Algorithm was terminated by the output function.
`-2`	Algorithm appears to be converging to a point that is not a root.
`-3`	Trust radius became too small.
`-4`	Line search cannot sufficiently decrease the residual along the current search direction.

`output` Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations
`algorithm`	Algorithm used.
`cgiterations`	Number of PCG iterations (large-scale algorithm only)
`stepsize`	Final step size taken (medium-scale algorithm only)

`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale problems, the first-order optimality is the infinity norm of the gradient $g = J^T F$ (see "Nonlinear Least-Squares" on page 4-10).

Options

Optimization options used by `fsolve`. Some options 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.You can use `optimset` to set or change the values of these fields in the options structure, `options`. See "Optimization Options" on page 5-9, for detailed information.

The `LargeScale` option specifies a preference for which algorithm to use. It is only a preference because certain conditions must be met to use the large-scale algorithm. For `fsolve`, 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 is used:

`LargeScale`	Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`. The default for `fsolve` is `'off'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`DerivativeCheck`	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.

`Jacobian`	If `'on'`, `fsolve` uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function. If `'off'`, `fsolve` approximates the Jacobian using finite differences.
`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`.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

`JacobMult`

Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form

```
W = jmfun(Jinfo,Y,flag,p1,p2,...)
```

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`, for example by

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for these parameters.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for a similar example.

`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 is 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 "Algorithm" on page 5-115).
`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.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`NonlEqnAlgorithm`	Choose Levenberg-Marquardt or Gauss-Newton over the trust region dogleg algorithm.
`LineSearchType`	Line search algorithm choice.

Examples

Example 1. This example finds a zero of the system of two equations and two unknowns:

$$
\begin{aligned}
2x_1 - x_2 &= e^{-x_1} \\
-x_1 + 2x_2 &= e^{-x_2}
\end{aligned}
$$

You 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 33 function evaluations, a zero is found.

```
                                         Norm of  First-order Trust-region
Iteration Func-count    f(x)           step   optimality      radius
    0          3       23535.6                  2.29e+004          1
    1          6       6001.72            1     5.75e+003          1
    2          9       1573.51            1     1.47e+003          1
    3         12       427.226            1           388          1
    4         15       119.763            1           107          1
    5         18       33.5206            1          30.8          1
    6         21       8.35208            1          9.05          1
    7         24       1.21394            1          2.26          1
    8         27      0.016329     0.759511         0.206        2.5
    9         30  3.51575e-006     0.111927       0.00294        2.5
   10         33  1.64763e-013   0.00169132     6.36e-007        2.5
Optimization terminated successfully:
 First-order optimality is less than options.TolFun

x =
    0.5671
    0.5671

fval =
```

```
  1.0e-006 *
      -0.4059
      -0.4059
```

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-009 *
    -0.1619    0.0776
     0.1161   -0.0469

exitflag =
     1
```

and the residual is close to zero.

```
sum(sum(Fval.*Fval))
ans =
     4.7915e-020
```

Notes

If the system of equations is linear, use the \ (the backslash operator; see `help slash`) for better speed and accuracy. For example, to find the solution to the following linear system of equations:

$$
\begin{aligned}
3x_1 + 11x_2 - 2x_3 &= 7 \\
x_1 + x_2 - 2x_3 &= 4 \\
x_1 - x_2 + x_3 &= 19
\end{aligned}
$$

You can formulate and solve the problem 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 Gauss-Newton, Levenberg-Marquardt, and large-scale methods are based on the nonlinear least-squares algorithms also used in `lsqnonlin`. Use one of these methods if the system may 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 might converge to a point that is not a solution of the system of equations (see “Limitations” and “Diagnostics” following).

Large-Scale Optimization. `fsolve`, with the `LargeScale` option set to `'on'` with `optimset`, uses the large-scale algorithm if possible. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1],[2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See “Trust-Region Methods for Nonlinear Minimization” on page 4-2 and “Preconditioned Conjugate Gradients” on page 4-5.

Medium-Scale Optimization. By default `fsolve` chooses the medium-scale algorithm and uses the trust-region dogleg method. The algorithm is a variant of the Powell dogleg method described in [8]. It is similar in nature to the algorithm implemented in [7].

Alternatively, you can select a Gauss-Newton method [3] with line-search, or a Levenberg-Marquardt method [4], [5], [6] with line-search. The choice of

algorithm is made by setting the `NonlEqnAlgorithm` option to `'dogleg'` (default), `'lm'`, or `'gn'`.

The default line search algorithm for the Levenberg-Marquardt and Gauss-Newton methods, i.e., the `LineSearchType` option 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 `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 "Standard Algorithms" chapter.

Diagnostics

Medium and Large-Scale Optimization. `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.

Medium-Scale Optimization. For the trust region dogleg method, `fsolve` stops if the step size becomes too small and it can make no more progress. `fsolve` gives this message:

```
The optimization algorithm can make no further progress:
 Trust region radius less than 10*eps
```

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. 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, might lead to a costly solution process for large problems.

Medium-Scale Optimization. The default trust region dogleg method can only be used when the system of equations is square, i.e., the number of equations equals the number of unknowns. For the Levenberg-Marquardt and Gauss-Newton methods, the system of equations need not be square.

See Also

`@(function_handle)`, `\`, `lsqcurvefit`, `lsqnonlin`, `optimset`, anonymous functions

References

[1] 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.

[2] 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.

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

[4] Levenberg, K., "A Method for the Solution of Certain Problems in Least-Squares," *Quarterly Applied Mathematics 2*, pp. 164-168, 1944.

[5] Marquardt, D., "An Algorithm for Least-squares Estimation of Nonlinear Parameters," *SIAM Journal Applied Mathematics*, Vol. 11, pp. 431-441, 1963.

[6] 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.

[7] Moré, J. J., B. S. Garbow, and K. E. Hillstrom, *User Guide for MINPACK 1*, Argonne National Laboratory, Rept. ANL-80-74, 1980.

[8] Powell, M. J. D., "A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations," *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, ed., Ch.7, 1970.

fzero

Purpose

Zero of a continuous function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[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 that `fzero` returns a value near a point where `fun` changes sign.

Note Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

`x = fzero(fun,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`[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.

"Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `fzero`. This section provides function-specific details for `fun` and `options`:

`fun` — The function whose zero is to be computed. `fun` is a function that accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fzero(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function.

```
x = fzero(@(x)sin(x*x),x0);
```

`options` — Optimization options. You can set or change the values of these options 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; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no warning.
`OutputFcn`	Specify a user-defined function that the optimization function calls at each iteration.
`TolX`	Termination tolerance on `x`.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `fzero`. This section provides function-specific details for `exitflag` and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

`1`	Function converged to a solution `x`.
`-1`	Algorithm was terminated by the output function.
`-3`	`NaN` or `Inf` function value was encountered during search for an interval containing a sign change.
`-4`	Complex function value was encountered during search for an interval containing a sign change.
`-5`	Algorithm might have converged to a singular point.

`output` Structure containing information about the optimization. The fields of the structure are

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations
`intervaliterations`	Number of iterations taken to find an interval
`iterations`	Number of zero-finding iterations
`message`	Exit message

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`, enter

```
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, enter

```
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
```

If fun is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function myfun defined by the following M-file function.

```
function f = myfun(x,a)
f = cos(a*x);
```

Note that myfun has an extra parameter a, so you cannot pass it directly to fzero. To optimize for a specific value of a, such as a = 2.

1. Assign the value to a.

   ```
   a = 2; % define parameter first
   ```

2. Call fzero with a one-argument anonymous function that captures that value of a and calls myfun with two arguments:

   ```
   x = fzero(@(x) myfun(x,a),0.1)
   ```

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 finds a point where the function changes sign. If the function is *continuous*, this is also a point where the function has a value near zero. If the function is not continuous, `fzero` may return values that are discontinuous points instead of zeros. For example, `fzero(@tan,1)` returns `1.5708`, a discontinuous point in `tan`.

Furthermore, 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 `@` (`function_handle`), `\`, `fminbnd`, `fsolve`, `optimset`, `roots`, anonymous functions

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

Multiplication with fundamental nullspace basis

Syntax

```
W = fzmult(A,V)
W = fzmult(A,V,'transpose')
[W,L,U,pcol,P]  = fzmult(A,V)
W = fzmult(A,V,TRANSPOSE,L,U,pcol,P)
```

Description

`W = fzmult(A,V)` computes the product `W` of matrix `Z` with matrix `V`, that is, `W = Z*V`, where `Z` is a fundamental basis for the nullspace of matrix `A`. `A` must be a sparse `m`-by-`n` matrix where `m < n`, `rank(A) = m`, and `rank(A(1:m,1:m)) = m`. `V` must be `p`-by-`q`, where `p = n-m`. If `V` is sparse `W` is sparse, else `W` is full.

`W = fzmult(A,V,'transpose')` computes the product of the transpose of the fundamental basis times `V`, that is, `W = Z'*V`. `V` must be `p`-by-`q`, where `q = n-m`. `fzmult(A,V)` is the same as `fzmult(A,V,[])`.

`[W,L,U,pcol,P] = fzmult(A,V)` returns the sparse LU-factorization of matrix `A(1:m,1:m)`, that is, `A1 = A(1:m,1:m)` and `P*A1(:,pcol) = L*U`.

`W = fzmult(A,V,transpose,L,U,pcol,P)` uses the precomputed sparse LU factorization of matrix `A(1:m,1:m)`, that is, `A1 = A(1:m,1:m)` and `P*A1(:,pcol) = L*U`. `transpose` is either `'transpose'` or `[]`.

The nullspace basis matrix `Z` is not formed explicitly. An implicit representation is used based on the sparse LU factorization of `A(1:m,1:m)`.

gangstr

Purpose Zero out “small” entries subject to structural rank

Syntax

```
A = gangstr(M,tol)
```

Description `A = gangstr(M,tol)` creates matrix `A` of full structural rank such that `A` is `M` except that elements of `M` that are relatively “small,” based on `tol`, are zeros in `A`. The algorithm decreases `tol`, if needed, until `sprank(A) = sprank(M)`. `M` must have at least as many columns as rows. Default `tol` is `1e-2`.

`gangstr` identifies elements of `M` that are relatively less than `tol` by first normalizing all the rows of `M` to have norm `1`. It then examines nonzeros in `M` in a columnwise fashion, replacing with zeros those elements with values of magnitude less than `tol*`(maximum absolute value in that column).

See Also `sprank`, `spy`

Purpose

Solve a linear programming problem

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

where 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 (the `LargeScale` option is set to `'off'` using `optimset`). The default large-scale algorithm and the simplex algorithm ignore any starting point.

`x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these 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`.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `linprog`. "Options" on page 5-127 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `linprog`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

`1`	Function converged to a solution `x`.
`0`	Number of iterations exceeded `options.MaxIter`.
`-2`	No feasible point was found.
`-3`	Problem is unbounded.
`-4`	`NaN` value was encountered during execution of the algorithm.
`-5`	Both primal and dual problems are infeasible.
`-7`	Search direction became too small. No further progress could be made.

`lambda` Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are:

`lower`	Lower bounds `lb`

	`upper`	Upper bounds `ub`
	`ineqlin`	Linear inequalities
	`eqlin`	Linear equalities
`output`	Structure containing information about the optimization. The fields of the structure are:	
	`algorithm`	Algorithm used
	`cgiterations`	The number of conjugate gradient iterations (large-scale algorithm only).
	`iterations`	Number of iterations
	`message`	Exit message

Options

Optimization options used by `linprog`. Some options apply to all algorithms, and others are only relevant when using the large-scale algorithm.You can use `optimset` to set or change the values of these fields in the options structure, `options`. See "Optimization Options" on page 5-9, for detailed information.:

`LargeScale`	Use large-scale algorithm when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-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'` (default) displays just the final output. At this time, the `'iter'` level only works with the large-scale algorithm.
`MaxIter`	Maximum number of iterations allowed.

Medium-Scale ALgorithm Only. These options are used by the medium-scale algorithm:

`Simplex`	If `'on'`, `linprog` uses the simplex algorithm. The simplex algorithm uses a built-in starting point, ignoring the starting point `x0` if supplied. The default is `'off'`. See "Simplex Algorithm" on page 3-36 for more information and an example.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

`TolFun`	Termination tolerance on the function value.

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 (Linear Interior Point Solver, [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 "Large-Scale Linear Programming" on page 4-13.

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]. The algorithm finds an initial feasible solution by first solving another linear programming problem.

Alternatively, you can use the simplex algorithm, described in "Simplex Algorithm" on page 3-36, by entering

```
options = optimset('LargeScale , 'off', 'Simplex', 'on')
```

and passing `options` as an input argument to `linprog`. The simplex algorithm returns a vertex optimal solution.

Note You cannot supply an initial point x0 for linprog with either the large-scale method or the medium-scale method using the simplex algorithm. In either case, if you pass in x0 as an input argument, linprog ignores x0 and computes its own initial point for the algorithm.

Diagnostics

Large-Scale Optimization. The first stage of the algorithm might involve some preprocessing of the constraints (see “Large-Scale Linear Programming” on page 4-13). Several possible conditions might occur that cause linprog to exit with an infeasibility message. In each case, the exitflag argument returned by linprog is 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 not to 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 of the algorithm begins until the stopping criteria are met. (See "Large-Scale Linear Programming" on page 4-13 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 is displayed, 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 is displayed, it is followed by one of the following six messages indicating that the dual, the primal, or both appear to be 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 problem 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` option 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

$$\min_x \frac{1}{2}\|F(x, xdata) - ydata\|_2^2 = \frac{1}{2}\sum_{i=1}^{m}(F(x, xdata_i) - ydata_i)^2$$

where *xdata* and *ydata* are vectors of length *m* 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,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 the vectors `ydata` and `xdata`.

`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 in `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 options specified in the structure `options`. Use `optimset` to set these options. Pass empty matrices for `lb` and `ub` if no bounds exist.

`[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] = lsqcurvefit(...)` returns the Jacobian of `fun` at the solution `x`.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `lsqcurvefit`. This section provides function-specific details for `fun` and `options`:

`fun` The function you want to fit. `fun` is a function that takes a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = lsqcurvefit(@myfun,x0,xdata,ydata)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x,xdata)
F = ...            % Compute function values at x
```

`fun` can also be a function handle for an anonymous function.

```
f = @(x,xdata)x(1)*xdata.^2+x(2)*sin(xdata),...
           'x','xdata';
x = lsqcurvefit(f,x0,xdata,ydata);
```

Note `fun` should return `fun(x,xdata)`, and not the sum-of-squares `sum((fun(x,xdata)-ydata).^2)`. The algorithm implicitly squares and sums `fun(x,xdata)-ydata`.

If the Jacobian can also be computed *and* the `Jacobian` option 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`, where `n` is the length of `x0`, 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` "Options" on page 5-138 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `lsqcurvefit`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	Function converged to a solution `x`.
2	Change in `x` was less than the specified tolerance.
3	Change in the residual was less than the specified tolerance.
4	Magnitude of search direction smaller than the specified tolerance
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
-1	Algorithm was terminated by the output function.
-2	Problem is infeasible: the bounds `lb` and `ub` are inconsistent.
-4	Optimization could not make further progress.

`lambda` Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are

`lower`	Lower bounds `lb`

	`upper`	Upper bounds `ub`
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`algorithm`	Algorithm used
	`cgiterations`	The number of PCG iterations (large-scale algorithm only)
	`stepsize`	The final step size taken (medium-scale algorithm only)
	`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient $g = J^T F$ (see "Nonlinear Least-Squares" on page 4-10).

Note The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See the examples below.

Options

Optimization options used by `lsqcurvefit`. Some options apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm.You can use `optimset` to set or change the values of these fields in the options structure `options`. See "Optimization Options" on page 5-9, for detailed information.

The `LargeScale` option specifies 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 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. 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'.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

DerivativeCheck	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
Diagnostics	Display diagnostic information about the function to be minimized.
Display	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' (default) displays just the final output.
Jacobian	If 'on', lsqcurvefit uses a user-defined Jacobian (defined in fun), or Jacobian information (when using JacobMult), for the objective function. If 'off', lsqcurvefit approximates the Jacobian using finite differences.
MaxFunEvals	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations allowed.
OutputFcn	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.
TypicalX	Typical x values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

`JacobMult`	Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form

```
W = jmfun(Jinfo,Y,flag,p1,p2,...)
```

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`, for example by

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for these parameters.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for a similar example.

`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 is 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 "Algorithm" on page 5-142).
`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.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

`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.

Examples

Given vectors of data *xdata* and *ydata*, suppose 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

$$\min_x \quad \frac{1}{2}\sum_{i=1}^{m}(F(x, xdata_i) - ydata_i)^2$$

where m is the length of xdata and ydata, the function F is defined by

```
F(x,xdata) = x(1)*xdata.^2 + x(2)*sin(xdata) + x(3)*xdata.^3
```

and the starting point is x0 = [10, 10, 10].

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 chooses the large-scale algorithm. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1], [2]. Each iteration involves the approximate solution of a large linear system using the method of

preconditioned conjugate gradients (PCG). See “Trust-Region Methods for Nonlinear Minimization” on page 4-2 and “Preconditioned Conjugate Gradients” on page 4-5.

Medium-Scale Optimization. `lsqcurvefit`, with the `LargeScale` option set to `'off'` with `optimset`, uses the Levenberg-Marquardt method with line-search [4], [5], [6]. Alternatively, a Gauss-Newton method [3] with line-search may be selected. You can choose the algorithm by setting the `LevenbergMarquardt` option with `optimset`. Setting `LevenbergMarquardt` to `'off'` (and `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., `LineSearchType` option set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. You can select a safeguarded cubic polynomial method by setting `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 “Standard Algorithms” chapter.

Diagnostics

Large-Scale Optimization. The large-scale method does not allow equal upper and lower bounds. For example, if `lb(2)==ub(2)`, `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` might 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 algorithm for `lsqcurvefit` does not solve underdetermined systems; it requires that the number of equations, i.e.,

the row dimension of F, be at least as great as the number of variables. In the underdetermined case, the medium-scale algorithm is used instead. See Table 2-4, Large-Scale Problem Coverage and Requirements, on page 2-42, 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, can 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.

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 `lsqcurvefit`.

See Also

`@(function_handle)`, `\`, `lsqlin`, `lsqnonlin`, `lsqnonneg`, `optimset`

References

[1] 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.

[2] 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.

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

[4] Levenberg, K., "A Method for the Solution of Certain Problems in Least-Squares," *Quarterly Applied Math. 2*, pp. 164-168, 1944.

[5] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal Applied Math.,* Vol. 11, pp. 431-441, 1963.

[6] 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.

lsqlin

Purpose

Solve the constrained linear least-squares problem

$$\min_x \frac{1}{2}\|Cx-d\|_2^2 \quad \text{such that} \quad \begin{aligned} A\cdot x &\le b \\ Aeq\cdot x &= beq \\ lb \le x &\le ub \end{aligned}$$

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 preceding problem 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 in `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`. Set `lb=[]` and `b=[]` if no bounds exist.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these 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.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to lsqlin. "Options" on page 5-148 provides the options values specific to lsqlin.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by lsqlin. This section provides function-specific details for exitflag, lambda, and output:

exitflag Integer identifying the reason the algorithm terminated. The following lists the values of exitflag and the corresponding reasons the algorithm terminated.

1	Function converged to a solution x.
3	Change in the residual was smaller than the specified tolerance
0	Number of iterations exceeded options.MaxIter.
-2	The problem is infeasible.
-4	Ill-conditioning prevents further optimization.
-7	Magnitude of search direction became too small. No further progress could be made.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields are	
	`lower`	Lower bounds `lb`
	`upper`	Upper bounds `ub`
	`ineqlin`	Linear inequalities
	`eqlin`	Linear equalities
`output`	Structure containing information about the optimization. The fields are	
	`iterations`	Number of iterations taken
	`algorithm`	Algorithm used
	`cgiterations`	Number of PCG iterations (large-scale algorithm only)
	`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient $g = C^TCx + C^Td$ (see "Nonlinear Least-Squares" on page 4-10).

Options

Optimization options used by `lsqlin`. You can set or change the values of these options using the `optimset` function. Some options apply to all algorithms, some are only relevant when you are using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm. See"Optimization Options" on page 5-9 for detailed information.

The `LargeScale` option specifies a preference for which algorithm to use. It is only a preference, because 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 is used:

`LargeScale`	Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`MaxIter`	Maximum number of iterations allowed.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

• JacobMult — Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form

```
W = jmfun(Jinfo,Y,flag,p1,p2,...)
```

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`, for example by

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.

If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for these parameters.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for a similar example.

`MaxPCGIter`

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`.

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for these parameters.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for a similar example.

`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.
`TolFun`	Termination tolerance on the function value.
`TolPCG`	Termination tolerance on the PCG iteration.

Examples

Find the least-squares solution to the overdetermined 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` produces

```
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, use `\`. For example, `x= A\b`.

Because the problem being solved is always convex, lsqlin will find a global, although not necessarily unique, 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, set the corresponding components of ub (or lb) 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 [1]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See "Trust-Region Methods for Nonlinear Minimization" on page 4-2 and "Preconditioned Conjugate Gradients" on page 4-5.

Medium-Scale Optimization. lsqlin, with the LargeScale option set to 'off' with optimset, or when linear inequalities or equalities are given, is based on quadprog, which uses an active set method similar to that described in [2]. It finds an initial feasible solution by first solving a linear programming problem. See "Quadratic Programming" on page 4-11 in the "Introduction to Algorithms" section.

Diagnostics

Large-Scale Optimization. The large-scale method 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, you must use the medium-scale algorithm 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 method, `lsqlin` warns that the matrices are converted to full.

```
Warning: This problem formulation not yet available for sparse
matrices.
Converting to full to solve.
```

When a problem is infeasible, `lsqlin` gives a warning:

```
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` option in `options`, are `'off'` and `'final'`; iterative output using `'iter'` is not available.

See Also

`\`, `lsqnonneg`, `quadprog`

References

[1] 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.

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

Purpose

Solve nonlinear least-squares (nonlinear data-fitting) problem

$$\min_x (f(x)) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + \ldots + f_m(x)^2$$

Syntax

```
x = lsqnonlin(fun,x0)
x = lsqnonlin(fun,x0,lb,ub)
x = lsqnonlin(fun,x0,lb,ub,options)
[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, you can restate this optimization problem 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 of 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. (The algorithm implicitly sums and squares `fun(x)`.)

`x = lsqnonlin(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range `lb <= x <= ub`.

`x = lsqnonlin(fun,x0,lb,ub,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options. Pass empty matrices for `lb` and `ub` if no bounds exist.

`[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`.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `lsqnonlin`. This section provides function-specific details for `fun` and `options`:

`fun` The function whose sum of squares is minimized. `fun` is a function that accepts a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = lsqnonlin(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x)
F = ...             % Compute function values at x
```

`fun` can also be a function handle for an anonymous function.

```
x = lsqnonlin(@(x)sin(x.*x),x0);
```

If the Jacobian can also be computed *and* the `Jacobian` option 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`, where `n` is the length of `x0`, 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` "Options" on page 5-160 provides the function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `lsqnonlin`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	Function converged to a solution `x`.
2	Change in `x` was less than the specified tolerance.
3	Change in the residual was less than the specified tolerance.
4	Magnitude of search direction was smaller than the specified tolerance.
0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
-1	Algorithm was terminated by the output function.
-2	Problem is infeasible: the bounds `lb` and `ub` are inconsistent.
-4	Line search could not sufficiently decrease the residual along the current search direction.

`lambda` Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields are

`lower`	Lower bounds `lb`
`upper`	Upper bounds `ub`

`output` Structure containing information about the optimization. The fields of the structure are

`iterations`	Number of iterations taken
`funcCount`	The number of function evaluations
`algorithm`	Algorithm used

`cgiterations`	Number of PCG iterations (large-scale algorithm only)
`stepsize`	The final step size taken (medium-scale algorithm only)
`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient $g = J^T F$ (see "Nonlinear Least-Squares" on page 4-10).

Note The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See the following example.

Options

Optimization options. You can set or change the values of these options using the `optimset` function. Some options apply to all algorithms, some are only relevant when you are using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm. See "Optimization Options" on page 5-9 for detailed information.

The `LargeScale` option specifies 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 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`. 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'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`DerivativeCheck`	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`Jacobian`	If `'on'`, `lsqnonlin` uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function. If `'off'`, `lsqnonlin` approximates the Jacobian using finite differences.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

JacobMult

Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form

```
W = jmfun(Jinfo,Y,flag,p1,p2,...)
```

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`, for example by

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for these parameters.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for a similar example.

`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 is 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 "Algorithm" on page 5-164).
`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.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

`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.

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` chooses the large-scale algorithm. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1], [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See “Trust-Region Methods for Nonlinear Minimization” on page 4-2 and “Preconditioned Conjugate Gradients” on page 4-5.

Medium-Scale Optimization. If you set the `LargeScale` option set to `'off'` with `optimset`, `lsqnonlin` uses the Levenberg-Marquardt method with line search [4], [5], [6]. Alternatively, you can select a Gauss-Newton method [3] with line search by setting the `LevenbergMarquardt` option. Setting `LevenbergMarquardt` to `'off'` (and `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., the `LineSearchType` option set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. You can select a safeguarded cubic polynomial method by setting the `LineSearchType` option 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 "Standard Algorithms" chapter.

Diagnostics

Large-Scale Optimization. The large-scale method does not allow equal upper and lower bounds. For example, if `lb(2)==ub(2)`, `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` might 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 underdetermined 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 underdetermined case, the medium-scale algorithm is used instead. (If bound constraints exist, a warning is issued and the problem is solved with the bounds ignored.) See Table 2-4, Large-Scale Problem Coverage and Requirements, on page 2-42, 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, can 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.

Medium-Scale Optimization. The medium-scale algorithm does not handle bound constraints.

Because the large-scale algorithm does not handle underdetermined systems and the medium-scale algorithm does not handle bound constraints, problems with both these characteristics cannot be solved by `lsqnonlin`.

See Also

`@(function_handle)`, `lsqcurvefit`, `lsqlin`, `optimset`

References

[1] 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.

[2] 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.

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

[4] Levenberg, K.,“A Method for the Solution of Certain Problems in Least-Squares,” *Quarterly Applied Math. 2*, pp. 164-168, 1944.

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

[6] 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.

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 options specified in the structure `options`. Use `optimset` to set these 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`.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `lsqnonneg`. This section provides function-specific details for `options`:

options — Use `optimset` to set or change the values of these fields in the options structure, `options`. See "Optimization Options" on page 5-9, for detailed information.

`Display`	Level of display. `'off'` displays no output; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `lsqnonneg`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` — Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

`1`	Function converged to a solution `x`.
`0`	Number of iterations exceeded `options.MaxIter`.

`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` — Structure containing information about the optimization. The fields are

`iterations`	Number of iterations taken
`algorithm`	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 overdetermined),

```
[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 20, `lsqlin` might be faster than `lsqnonneg`; otherwise `lsqnonneg` is generally more efficient.

See Also

`\`, `lsqlin`, `optimset`

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least-Squares Problems*, Prentice-Hall, Chapter 23, p. 161, 1974.

optimget

Purpose

Get optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified option in the optimization options structure `options`. You need to type only enough leading characters to define the option name uniquely. Case is ignored for option names.

`val = optimget(options,'param',default)` returns `default` if the specified option 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` option in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` option in the structure called `my_options` (as in the previous example) except that if the `Display` option is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset`

optimset

Purpose

Create or edit optimization options 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 structure called `options`, in which the specified options (`param`) have specified values. Any unspecified options are set to `[]` (options with value `[]` indicate to use the default value for that option when you pass `options` to the optimization function). It is sufficient to type only enough leading characters to define the option name uniquely. Case is ignored for option names.

`optimset` with no input or output arguments displays a complete list of options 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 option names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified options with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any options in `newopts` with nonempty values overwrite the corresponding old options in `oldopts`.

Options

For more information about individual options, see the reference pages for the optimization functions that use these options. "Optimization Options" on page 5-9 provides descriptions of these options and which functions use them.

In the following lists, values in { } denote the default value; some options have different defaults for different optimization functions and so no values are shown in { }.

You can also view the optimization options and defaults by typing `optimset` at the command line.

Optimization options used by both large-scale and medium-scale algorithms:

`DerivativeCheck`	`'on'` | {`'off'`}
`Diagnostics`	`'on'` | {`'off'`}
`Display`	`'off'` | `'iter'` | `'final'` | `'notify'`
`FunValCheck`	{`'off'`} | `'on'`
`GradObj`	`'on'` | {`'off'`}
`Jacobian`	`'on'` | {`'off'`}
`LargeScale`	`'on'` | `'off'`. The default for `fsolve` is `'off'`. The default for all other functions that provide a large-scale algorithm is `'on'`.
`MaxFunEvals`	Positive integer
`MaxIter`	Positive integer
`OutputFcn`	Specify a user-defined function that an opimization function calls at each iteration. See "Output Function" on page 5-15.
`TolCon`	Positive scalar
`TolFun`	Positive scalar
`TolX`	Positive scalar
`TypicalX`	Vector of all ones

Optimization options used by large-scale algorithms only:

`Hessian`	`'on'` | {`'off'`}
`HessMult`	Function | {`[]`}
`HessPattern`	Sparse matrix | {sparse matrix of all ones}

`InitialHessMatrix`	{`'identity'`} | `'scaled-identity'` | `'user-supplied'`
`InitialHessType`	scalar | vector | {`[]`}
`JacobMult`	Function | {`[]`}
`JacobPattern`	Sparse matrix |{sparse matrix of all ones}
`MaxPCGIter`	Positive integer | {the greater of 1 and `floor(n/2))`} where n is the number of elements in `x0`, the starting point
`PrecondBandWidth`	Positive integer | {0} | `Inf`
`TolPCG`	Positive scalar | {`0.1`}

Optimization options used by medium-scale algorithms only:

`BranchStrategy`	`'mininfeas'` | {`'maxinfeas'`}
`DiffMaxChange`	Positive scalar | {`1e 1`}
`DiffMinChange`	Positive scalar | {`1e 8`}
`GoalsExactAchieve`	Positive scalar integer | {`0`}
`GradConstr`	`'on'` | {`'off'`}
`HessUpdate`	{`'bfgs'`} | `'dfp'` | `'steepdesc'`
`LevenbergMarquardt`	`'on'` | {`'off'`}
`LineSearchType`	`'cubicpoly'` | {`'quadcubic'`}
`MaxNodes`	Positive scalar | {`1000*NumberOfVariables`}
`MaxRLPIter`	Positive scalar | {`100*NumberOfVariables`}
`MaxSQPIter`	Positive integer
`MaxTime`	Positive scalar | {`7200`}
`MeritFunction`	`'singleobj'` | {`'multiobj'`}
`MinAbsMax`	Positive scalar integer | {`0`}
`NodeDisplayInterval`	Positive scalar | {`20`}
`NodeSearchStrategy`	`'df'` | {`'bn'`}

`NonlEqnAlgorithm`	{`'dogleg'`} | `'lm'` | `'gn'`, where `'lm'` is Levenburg-Marquardt and `'gn'` is Gauss-Newton.
`Simplex`	When you set `'Simplex'` to `'on'` and `'LargeScale'` to `'off'`, `fmincon` uses the simplex algorithm to solve a contrained linear programming problem.
`TolRLPFun`	Positive scalar | {`1e-6`}
`TolXInteger`	Positive scalar | {`1e-8`}

Examples

This statement creates an optimization options structure called `options` in which the `Display` option is set to `'iter'` and the `TolFun` option 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` option and storing new values in `optnew`.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure `options` that contains all the option 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`

quadprog

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 preceding problem while additionally satisfying the equality constraints `Aeq*x = beq`.

`x = quadprog(H,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 in the range `lb <= x <= ub`.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0)` sets the starting point to `x0`.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these 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`.

Input Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments passed in to `quadprog`. "Options" on page 5-178 provides function-specific details for the `options` values.

Output Arguments

"Function Arguments" on page 5-5 contains general descriptions of arguments returned by `quadprog`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	Function converged to a solution `x`.
3	Change in the objective function value was smaller than the specified tolerance.
4	Local minimizer was found.
0	Number of iterations exceeded `options.MaxIter`.
-2	Problem is infeasible.
-3	Problem is unbounded.
-4	Current search direction was not a direction of descent. No further progress could be made.
-7	Magnitude of search direction became too small. No further progress could be made.

`lambda` Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields are

	`lower`	Lower bounds `lb`
	`upper`	Upper bounds `ub`
	`ineqlin`	Linear inequalities
	`eqlin`	Linear equalities
`output`	Structure containing information about the optimization. The fields are	
	`iterations`	Number of iterations taken
	`algorithm`	Algorithm used
	`cgiterations`	Number of PCG iterations (large-scale algorithm only)
	`firstorderopt`	Measure of first-order optimality (large-scale algorithm only) For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in "Box Constraints" on page 4-7, and `g` is the gradient. For large scale problems with linear equalities only, the first-order optimality is the 2-norm of the scaled residual (`z = M\r`) of the reduced preconditioned conjugate gradient calculation. See "Algorithm" on page 4-5 in "Preconditioned Conjugate Gradients," and also "Linearly Constrained Problems" on page 4-7.

Options

Optimization options. Use `optimset` to set or change the values of these options. Some options apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm. See "Optimization Options" on page 5-9 for detailed information.

The option to set an algorithm preference:

`LargeScale`	Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`. `'on'` is only a preference. If 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 is used.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`MaxIter`	Maximum number of iterations allowed.
`TypicalX`	Typical `x` values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

HessMult	Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product H*Y without actually forming H. The function is of the form

```
W = hmfun(Hinfo,Y,p1,p2,...)
```

where Hinfo and possibly the additional parameters p1,p2,... contain the matrices used to compute H*Y.

The first argument must be the same as the third argument returned by the objective function fun, for example by

```
[f,g,Hinfo] = fun(x)
```

Y is a matrix that has the same number of rows as there are dimensions in the problem. W = H*Y although H is not formed explicitly. fminunc uses Hinfo to compute the preconditioner. The optional parameters p1, p2, ... can be any additional parameters needed by hmfun. See “Avoiding Global Variables via Anonymous and Nested Functions” on page 2-19 for information on how to supply values for the parameters.

Note 'Hessian' must be set to 'on' for Hinfo to be passed from fun to hmfun.

See “Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints” on page 2-59 for an example.

MaxPCGIter	Y is a matrix that has the same number of rows as there are dimensions in the problem. W = H*Y although H is not formed explicitly. fminunc uses Hinfo to compute the preconditioner. The optional parameters p1, p2, ... can be any additional parameters needed by hmfun. See "Avoiding Global Variables via Anonymous and Nested Functions" on page 2-19 for information on how to supply values for the parameters. **Note** 'Hessian' must be set to 'on' for Hinfo to be passed from fun to hmfun. See "Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 2-59 for an example.
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.
TolFun	Termination tolerance on the function value. TolFun is used as the exit criterion for problems with simple lower and upper bounds (lb, ub).
TolPCG	Termination tolerance on the PCG iteration. TolPCG is used as the exit criterion for problems with only equality constraints (Aeq, beq).
TolX	Termination tolerance on x.

Examples

Find values of x that minimize

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1 x_2 - 2x_1 - 6x_2$$

subject to

$$\begin{aligned} x_1 + x_2 &\le 2 \\ -x_1 + 2x_2 &\le 2 \\ 2x_1 + x_2 &\le 3 \\ 0 \le x_1, \quad 0 &\le x_2 \end{aligned}$$

First, note that this function can be written in matrix notation as

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

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 the 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. IBy default, `quadprog` uses the large-scale algorithm if you specify the feasible region using, but not both, of the following types of constraints:

- Upper and lower bounds constraints
- Linear equality constraints, in which the columns of the constraint matrix `Aeq` are linearly independent. `Aeq` is typically sparse.

You cannot use inequality constraints with the large-scale algorithm. If the preceding conditions are not met, `quadprog` reverts to the medium-scale algorithm.

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, 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 option is 'off'), and the x returned is not a solution but a direction of negative curvature with respect to H.

For problems with simple lower and upper bounds (lb, ub), quadprog exits based on the value of TolFun. For problems with only equality constraints (Aeq, beq), the exit is based on TolPCG. Adjust TolFun and TolPCG to affect your results. TolX is used by both types of problems.

Algorithm

Large-Scale Optimization. The large-scale algorithm is a subspace trust-region method based on the interior-reflective Newton method described in [1]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See "Trust-Region Methods for Nonlinear Minimization" on page 4-2 and "Preconditioned Conjugate Gradients" on page 4-5.

Medium-Scale Optimization. quadprog uses an active set method, which is also a projection method, similar to that described in [2]. It finds an initial feasible solution by first solving a linear programming problem. This method is discussed in the "Standard Algorithms" chapter.

Diagnostics

Large-Scale Optimization. The large-scale method does 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, can 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 option 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 that a minimum was not found); when a finite solution does exist, quadprog might only give local minima, because the problem might 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 occurs, the medium-scale algorithm is called instead. See Table 2-4, Large-Scale Problem Coverage and Requirements, on page 2-42, for more information on what problem formulations are covered and what information must be provided.

References

[1] 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.

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

Index

Symbols
||C times x minus d|| squared 1-4

A
active constraints
  linprog example 5-129
  lsqlin example 5-154
  quadprog example 5-184
active set method
  fmincon medium-scale algorithm 5-63
  linprog medium-scale algorithm 5-129
  lsqlin medium-scale algorithm 5-154
  quadprog medium-scale algorithm 5-185
  sequential quadratic programming (SQP) 3-32
attainment factor 5-42
axis crossing. *See* zero of a function

B
banana function 3-4
BFGS formula 3-6
  fmincon medium-scale algorithm 5-63
  fminunc medium-scale algorithm 5-89
bintprog 5-26
bisection search 5-122
bound constraints, large-scale 4-7
box constraints. *See* bound constraints

C
centering parameter 4-15
CG. *See* conjugate gradients
complementarity conditions 4-14
complex variables 5-143, 5-165
conjugate gradients 4-3
constrained minimization 5-50
  large-scale example 2-54, 2-58
  medium-scale example 2-11
constraints
  linear 4-7, 5-63, 5-73
  positive 2-18
continuous derivative
  gradient methods 3-4
convex problem 3-27
curve-fitting 5-134
  categories 2-5
  functions that apply 5-3

D
data-fitting 5-134
  categories 2-5
  functions that apply 5-3
dense columns, constraint matrix 4-15
DFP formula 5-89
direction of negative curvature 4-3
discontinuities 2-93
discontinuous problems 5-78, 5-90
discrete variables 2-94
dual problem 4-13
duality gap 4-14

E
ε-constraint method 3-45
equality constraints
  dense columns 2-73
  medium-scale example 2-17
equality constraints inconsistent warning,
    quadprog 5-186
equality constraints, linear
  large-scale 4-7

equation solving
  categories 2-5
  functions that apply 5-2
error, `Out of memory.` 2-49

F

f(x) = one half ||C times x + d|| squared 4-12
feasibility conditions 4-14
feasible point, finding 3-34
`fgoalattain` **5-33**
  example 2-37
fixed variables 4-16
fixed-step ODE solver 2-33
`fminbnd` **5-45**
`fmincon` **5-50**
  large-scale example 2-54, 2-58
  medium-scale example 2-11
`fminimax` **5-65**
  example 2-33
`fminsearch` **5-75**
`fminunc` **5-80**
  large-scale example 2-51
  medium-scale example 2-10
  warning messages 2-93
`fseminf` **5-92**
`fsolve` **5-106**
  large-scale Jacobian 2-44
  medium-scale analytic Jacobian 2-23
  medium-scale finite difference Jacobian 2-26
`fsolve` medium-scale default 5-115
function arguments 5-5
function discontinuities 2-93
functions
  grouped by category 5-2
`fzero` **5-118**
`fzmult` **5-123**

G

`gangstr` **5-124**
Gauss-Newton method (large-scale)
  nonlinear least-squares 4-10
Gauss-Newton method (medium-scale)
  implementation, nonlinear equations 3-25
  implementation, nonlinear least squares 3-21
  least-squares optimization 3-18
  solving nonlinear equations 3-23
global minimum 2-92
goal attainment 3-47, 5-33
  example 2-37
`goaldemo` 5-41
golden section search 5-48
gradient checking, analytic 2-16
gradient examples 2-14
gradient function 2-7
gradient methods
  continuous first derivative 3-4
  quasi-Newton 3-6
  unconstrained optimization 3-4

H

`Hessian modified` message 3-31
`Hessian modified twice` message 3-31
Hessian sparsity structure 2-53
Hessian update 3-10, 3-30
Hessian updating methods 3-6

I

inconsistent constraints 5-132
indefinite problems 5-186
`infeasible` message 3-32
infeasible optimization problems 2-93
infeasible problems 5-64

infeasible solution warning
  `linprog` 5-132
  `quadprog` 5-185
inline objects 2-90
input arguments 5-5
integer variables 2-94
interior-point linear programming 4-13
introduction to optimization 3-3
iterative display 2-82

J

Jacobian
  analytic 2-23
  finite difference 2-26
  large-scale nonlinear equations 2-44
Jacobian sparsity pattern 2-47

K

Kuhn-Tucker equations 3-27

L

Lagrange multipliers
  large-scale linear programming 4-16
large-scale functionality coverage 2-41
large-scale methods 4-1
  demos 5-3
  examples 2-40
least squares 3-19
  categories 2-5
  functions that apply 5-3
Levenberg-Marquardt method 3-19
  `lsqcurvefit` medium-scale default 5-143
  `lsqnonlin` medium-scale default 5-164
  search direction 3-20
line search
  `fminunc` medium-scale default 5-89
  `fsolve` medium-scale default 5-115
  `lsqcurvefit` medium-scale default 5-143
  `lsqnonlin` medium-scale default 5-164
  unconstrained optimization 3-8
line search strategy 2-7
linear constraints 4-7, 5-63, 5-73
linear equations solve 5-115
linear least squares
  constrained 5-146
  large-scale algorithm 4-12
  large-scale example 2-70
  nonnegative 5-167
  unconstrained 5-154
linear programming 5-125
  implementation 3-35
  large-scale algorithm 4-13
  large-scale example 2-72, 2-73
  problem 3-3
`linprog` **5-125**
  large-scale example 2-72, 2-73
LIPSOL 4-13
lower bounds 2-13
`lsqcurvefit` **5-134**
`lsqlin` **5-146**
  large-scale example 2-70
`lsqnonlin` **5-156**
  convergence 2-95
  large-scale example 2-47
  medium-scale example 2-30
`lsqnonneg` **5-167**

M

maximization 2-18
medium-scale methods 3-1

demos 5-4
Mehrotra's predictor-corrector algorithm 4-13, 4-14
merit function 3-35
minimax examples 2-33
minimax problem, solving 5-65
minimization
categories 2-3
functions that apply 5-2
multiobjective optimization 3-41, 5-33
examples 2-27

N

NCD. *See* Nonlinear Control Design
negative curvature direction 4-3, 4-5
negative definite problems 5-186
Nelder and Mead 3-4
Newton direction
approximate 4-3
Newton's method
systems of nonlinear equations 3-23
unconstrained optimization 3-4
`no update` message 3-32
nonconvex problems 5-186
noninferior solution 3-42
Nonlinear Control Design (NCD) Blockset 2-33
nonlinear data-fitting 5-156
nonlinear equations
Newton's method 3-23
nonlinear equations (large-scale)
example with Jacobian 2-44
solving 5-106
nonlinear equations (medium-scale) 3-23
analytic Jacobian example 2-23
finite difference Jacobian example 2-26
Gauss-Newton method 3-23
solving 5-106
trust-region dogleg method 3-23
nonlinear least squares 3-21, 5-134, 5-156
large-scale algorithm 4-10
large-scale example 2-47
nonlinear programming 3-3
normal equations 4-10, 4-12

O

objective function 2-3
return values 2-95
optimality conditions linear programming 4-14
`optimget` **5-171**
optimization
functions by category 5-2
getting to a global minimum 2-92
handling infeasibility 2-93
helpful hints 2-92
introduction 3-3
objective function return values 2-95
troubleshooting 2-92
unconstrained 3-4
optimization parameters structure 2-76, 5-171, 5-172
`optimset` **5-172**
options parameters
descriptions 5-9
possible values 5-173
utility functions 5-3
`Out of memory.` error 2-49
output arguments 5-5
output display 2-79
output function 2-85
output headings 2-82
large-scale algorithms 2-82
medium-scale algorithms 2-79

P

PCG. *See* preconditioned conjugate gradients
preconditioned conjugate gradients 4-3, 4-5, 4-15
  algorithm 4-5
preconditioner 2-46, 4-5
  banded 2-54
predictor-corrector algorithm 4-14
preprocessing
  linear programming 4-13, 4-16
primal problem 4-13
primal-dual algorithm 4-14
primal-dual interior-point 4-13
projection method
  `quadprog` medium-scale algorithm 5-185
  sequential quadratic programming (SQP) 3-32

Q

`quadprog` **5-176**
  large-scale example 2-63
quadratic programming 3-3, 5-63, 5-176
  large-scale algorithm 4-11
  large-scale example 2-63
quasi-Newton method
  implementation 3-10
quasi-Newton methods 3-6
  `fminunc` medium-scale algorithm 5-89
  unconstrained optimization 3-6

R

reflective line search 4-11
reflective steps 4-8, 4-9
residual 3-17
revised simplex algorithm 3-36
Rosenbrock’s function 3-4

S

S 4-10
sampling interval 5-99
secular equation 4-3
semi-infinite constraints 5-92
Sherman-Morrison formula 4-15
signal processing example 2-36
simple bounds 2-13
simplex search 5-78
  unconstrained optimization 3-4
Simulink, multiobjective example 2-27
singleton rows 4-16
solving nonlinear systems of equations 3-23
sparsity pattern Jacobian 2-47
sparsity structure, Hessian 2-53
SQP method 3-28, 3-32, 5-63
steepest descent 5-89
stopping criteria, large-scale linear programming 4-16
structural rank 4-16
subspace
  determination of 4-3
subspace, two-dimensional 4-3
systems of nonlinear equations
  solving 5-106

T

trust region 4-2
trust-region dogleg method (medium-scale)
  implementation for nonlinear equations 3-25
  systems of nonlinear equations 3-23
two-dimensional subspace 4-3

U

unbounded solutions warning

linprog 5-132
quadprog 5-186
unconstrained minimization 5-75, 5-80
large-scale example 2-51
medium-scale example 2-10
one dimensional 5-45
unconstrained optimization 3-4
upper bounds 2-13

V

variable-step ODE solver 2-33

W

warning
equality constraints inconsistent, quadprog 5-186
infeasible solution, linprog 5-132
infeasible solution, quadprog 5-185
stuck at minimum, fsolve 5-116
unbounded solutions, linprog 5-132
unbounded solutions, quadprog 5-186
warnings displayed 2-94
weighted sum strategy 3-43

Z

zero curvature direction 4-5
zero finding 5-106
zero of a function, finding 5-118