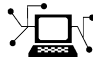
Optimization Toolbox™ 4

User's Guide

MATLAB®

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Optimization Toolbox™ User's Guide

© COPYRIGHT 1990–2008 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.

Trademarks

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

Patents

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

Revision 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)
October 2004	Online only	Revised for Version 3.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.0.3 (Release 14SP3)
March 2006	Online only	Revised for Version 3.0.4 (Release 2006a)
September 2006	Sixth printing	Revised for Version 3.1 (Release 2006b)
March 2007	Seventh printing	Revised for Version 3.1.1 (Release 2007a)
September 2007	Eighth printing	Revised for Version 3.1.2 (Release 2007b)
March 2008	Online only	Revised for Version 4.0 (Release 2008a)
October 2008	Online only	Revised for Version 4.1 (Release 2008b)

Acknowledgments

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

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 Dean of Faculty of Mathematics and Professor of Combinatorics and Optimization at University of Waterloo.

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 Professor of Computational and Applied Mathematics on the faculty of the Keck Center for Interdisciplinary Bioscience Training at Rice University.

Dr. Zhang has published over 50 technical papers in the areas of interior-point methods for linear programming and computation mathematical programming.

Contents

1 Getting Started

Product Overview **1-2**
Introduction **1-2**
Optimization Functions **1-2**
Optimization Tool GUI **1-3**

Example: Nonlinear Constrained Minimization **1-4**
Problem Formulation: Rosenbrock's Function **1-4**
Defining the Problem in Toolbox Syntax **1-5**
Running the Optimization **1-7**
Interpreting the Result **1-12**

2 Optimization Overview

Introduction to Optimization Toolbox Solvers **2-2**

Writing Objective Functions **2-4**
Writing Objective Functions **2-4**
Jacobians of Vector and Matrix Objective Functions **2-6**
Anonymous Function Objectives **2-9**
Maximizing an Objective **2-9**

Writing Constraints **2-11**
Types of Constraints **2-11**
Bound Constraints **2-12**
Linear Inequality Constraints **2-13**
Linear Equality Constraints **2-13**
Nonlinear Constraints **2-14**
An Example Using All Types of Constraints **2-15**

Passing Extra Parameters **2-17**
Anonymous Functions **2-17**
Nested Functions **2-19**
Global Variables **2-20**

Choosing a Solver **2-21**
Problems Handled by Optimization Toolbox Functions ... **2-21**
Optimization Decision Table **2-24**

Solver Inputs and Outputs **2-27**
Iterations and Function Counts **2-27**
First-Order Optimality Measure **2-28**
Tolerances and Stopping Criteria **2-31**
Lagrange Multiplier Structures **2-32**
Output Structures **2-33**
Output Functions **2-33**

Default Options Settings **2-41**
Introduction **2-41**
Changing the Default Settings **2-41**
Large-Scale vs. Medium-Scale Algorithms **2-45**

Displaying Iterative Output **2-47**
Introduction **2-47**
Most Common Output Headings **2-47**
Function-Specific Output Headings **2-48**

Typical Problems and How to Deal with Them **2-54**

Local vs. Global Optima **2-57**
What Are Local and Global Optima? **2-57**
Basins of Attraction **2-57**
Searching For Global Optima **2-59**

Reference **2-62**

3 Optimization Tool

Getting Started with the Optimization Tool **3-2**
Introduction **3-2**
Opening the Optimization Tool **3-2**
Steps for Using the Optimization Tool **3-5**

Running a Problem in the Optimization Tool **3-6**
Introduction **3-6**
Pausing and Stopping the Algorithm **3-7**
Viewing Results **3-7**
Final Point **3-7**
Starting a New Problem **3-8**
Closing the Optimization Tool **3-9**

Specifying Certain Options **3-10**
Plot Functions **3-10**
Output function **3-11**
Display to Command Window **3-11**

Getting Help in the Optimization Tool **3-13**
Quick Reference **3-13**
Additional Help **3-13**

Importing and Exporting Your Work **3-14**
Exporting to the MATLAB Workspace **3-14**
Importing Your Work **3-16**
Generating an M-File **3-16**

Optimization Tool Examples **3-18**
About Optimization Tool Examples **3-18**
Optimization Tool with the fmincon Solver **3-18**
Optimization Tool with the lsqlin Solver **3-22**

4 Using Optimization Toolbox Solvers

Optimization Theory Overview **4-2**

Unconstrained Nonlinear Optimization **4-3**
Definition **4-3**
Large Scale fminunc Algorithm **4-3**
Medium Scale fminunc Algorithm **4-6**
fminsearch Algorithm **4-11**

Unconstrained Nonlinear Optimization Examples **4-14**
Example: fminunc Unconstrained Minimization **4-14**
Example: Nonlinear Minimization with Gradient and Hessian **4-16**
Example: Nonlinear Minimization with Gradient and Hessian Sparsity Pattern **4-17**

Constrained Nonlinear Optimization **4-20**
Definition **4-20**
fmincon Trust Region Reflective Algorithm **4-20**
fmincon Active Set Algorithm **4-26**
fmincon Interior Point Algorithm **4-35**
fminbnd Algorithm **4-39**
fseminf Problem Formulation and Algorithm **4-39**

Constrained Nonlinear Optimization Examples **4-44**
Example: Nonlinear Inequality Constraints **4-44**
Example: Bound Constraints **4-46**
Example: Constraints With Gradients **4-47**
Example: Constrained Minimization Using fmincon's Interior-Point Algorithm With Analytic Hessian **4-50**
Example: Equality and Inequality Constraints **4-57**
Example: Nonlinear Minimization with Bound Constraints and Banded Preconditioner **4-58**
Example: Nonlinear Minimization with Equality Constraints **4-62**
Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints **4-64**
Example: One-Dimensional Semi-Infinite Constraints ... **4-68**
Example: Two-Dimensional Semi-Infinite Constraint **4-70**

Linear Programming **4-74**
Definition **4-74**
Large Scale Linear Programming **4-74**
Active-Set Medium-Scale linprog Algorithm **4-78**
Medium-Scale linprog Simplex Algorithm **4-82**

Linear Programming Examples **4-86**
Example: Linear Programming with Equalities and Inequalities **4-86**
Example: Linear Programming with Dense Columns in the Equalities **4-87**

Quadratic Programming **4-90**
Definition **4-90**
Large-Scale quadprog Algorithm **4-90**
Medium-Scale quadprog Algorithm **4-95**

Quadratic Programming Examples **4-100**
Example: Quadratic Minimization with Bound Constraints **4-100**
Example: Quadratic Minimization with a Dense but Structured Hessian **4-102**

Binary Integer Programming **4-108**
Definition **4-108**
bintprog Algorithm **4-108**

Binary Integer Programming Example **4-111**
Example: Investments with Constraints **4-111**

Least Squares (Model Fitting) **4-116**
Definition **4-116**
Large-Scale Least Squares **4-117**
Levenberg-Marquardt Method **4-121**
Gauss-Newton Method **4-122**

Least Squares (Model Fitting) Examples **4-126**
Example: Using lsqnonlin With a Simulink Model **4-126**
Example: Nonlinear Least-Squares with Full Jacobian Sparsity Pattern **4-131**

Example: Linear Least-Squares with Bound Constraints 4-133
Example: Jacobian Multiply Function with Linear Least Squares 4-134
Example: Nonlinear Curve Fitting with lsqcurvefit 4-139

Multiobjective Optimization 4-141
Definition 4-141
Algorithms 4-142

Multiobjective Optimization Examples 4-147
Example: Using fminimax with a Simulink Model 4-147
Example: Signal Processing Using fgoalattain 4-150

Equation Solving 4-154
Definition 4-154
Trust-Region Dogleg Method 4-155
Trust-Region Reflective fsolve Algorithm 4-157
Levenberg-Marquardt Method 4-160
Gauss-Newton Method 4-160
\ Algorithm 4-161
fzero Algorithm 4-161

Equation Solving Examples 4-162
Example: Nonlinear Equations with Analytic Jacobian 4-162
Example: Nonlinear Equations with Finite-Difference Jacobian 4-165
Example: Nonlinear Equations with Jacobian 4-166
Example: Nonlinear Equations with Jacobian Sparsity Pattern 4-169

Selected Bibliography 4-172

5 Parallel Computing for Optimization

Parallel Computing in Optimization Toolbox Functions 5-2

Parallel Optimization Functionality 5-2
Parallel Estimation of Gradients 5-2
Nested Parallel Functions 5-4

Using Parallel Computing with fmincon, fgoalattain, and fminimax **5-5**
Using Parallel Computing with Multicore Processors 5-5
Using Parallel Computing with a Multiprocessor Network 5-6
Testing Parallel Computations 5-7

Improving Performance with Parallel Computing **5-8**
Factors That Affect Speed 5-8
Factors That Affect Results 5-8
Searching for Global Optima 5-9

6 External Interface

ktrlink: An Interface to KNITRO Libraries **6-2**
What Is ktrlink? 6-2
Installation and Configuration 6-2
Example Using ktrlink 6-4
Setting Options 6-7
Sparse Matrix Considerations 6-9

7 Argument and Options Reference

Function Arguments **7-2**
Input Arguments 7-2
Output Arguments 7-5

Optimization Options **7-7**
Options Structure 7-7
Output Function 7-17

Plot Functions 7-26

8 Function Reference

Minimization **8-2**

Equation Solving **8-2**

Least Squares (Curve Fitting) **8-3**

GUI **8-3**

Utilities **8-4**

9 Functions — Alphabetical List

A Examples

Constrained Nonlinear Examples **A-2**

Least Squares Examples **A-2**

Unconstrained Nonlinear Examples **A-2**

Linear Programming Examples **A-3**

Quadratic Programming Examples **A-3**

Binary Integer Programming Examples **A-3**

Multiobjective Examples **A-3**

Equation Solving Examples **A-3**

Index

1

Getting Started

- "Product Overview" on page 1-2
- "Example: Nonlinear Constrained Minimization" on page 1-4

Product Overview

In this section...
"Introduction" on page 1-2
"Optimization Functions" on page 1-2
"Optimization Tool GUI" on page 1-3

Introduction

Optimization Toolbox software extends the capability of the MATLAB® numeric computing environment. The software includes functions for many types of optimization including

- Unconstrained nonlinear minimization
- Constrained nonlinear minimization, including semi-infinite minimization problems
- Quadratic and linear programming
- Nonlinear least-squares and curve fitting
- Constrained linear least squares
- Sparse and structured large-scale problems, including linear programming and constrained nonlinear minimization
- Multiobjective optimization, including goal attainment problems and minimax problems

The toolbox also includes functions for solving nonlinear systems of equations.

Optimization Functions

Most 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 Optimization Toolbox software by writing your own M-files, or by using the software in combination with other toolboxes, or with the MATLAB or Simulink® environments.

Optimization Tool GUI

Optimization Tool (`optimtool`) is a graphical user interface (GUI) for selecting a toolbox function, specifying optimization options, and running optimizations. It provides a convenient interface for all optimization routines, including those from Genetic Algorithm and Direct Search Toolbox™ software, which is licensed separately.

Optimization Tool makes it easy to

- Define and modify problems quickly
- Use the correct syntax for optimization functions
- Import and export from the MATLAB workspace
- Generate code containing your configuration for a solver and options
- Change parameters of an optimization during the execution of certain Genetic Algorithm and Direct Search Toolbox functions

Example: Nonlinear Constrained Minimization

In this section...

"Problem Formulation: Rosenbrock's Function" on page 1-4
"Defining the Problem in Toolbox Syntax" on page 1-5
"Running the Optimization" on page 1-7
"Interpreting the Result" on page 1-12

Problem Formulation: Rosenbrock's Function

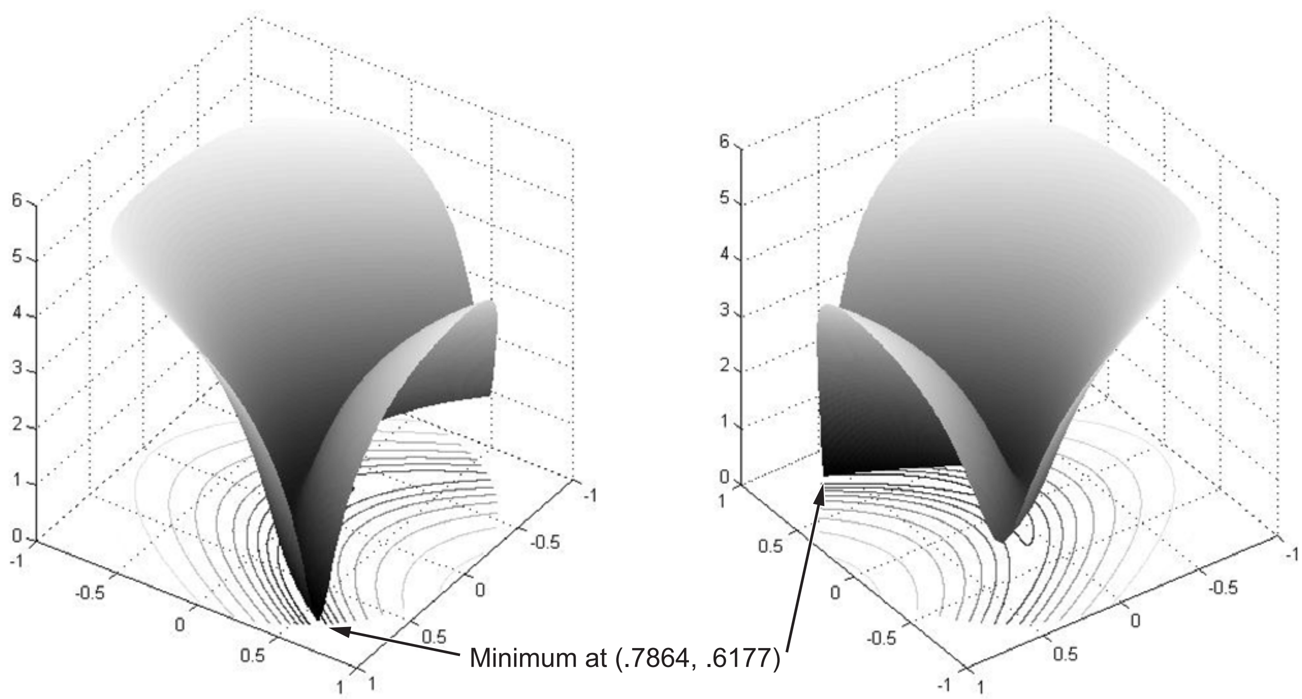
Consider the problem of minimizing Rosenbrock's function

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

over the *unit disk*, i.e., the disk of radius 1 centered at the origin. In other words, find x that minimizes the function $f(x)$ over the set $x_1^2 + x_2^2 \le 1$. This problem is a minimization of a nonlinear function with a nonlinear constraint.

Note Rosenbrock's function is a standard test function in optimization. It has a unique minimum value of 0 attained at the point (1,1). Finding the minimum is a challenge for some algorithms since it has a shallow minimum inside a deeply curved valley.

Here are two view of Rosenbrock's function in the unit disk. The vertical axis is log-scaled; in other words, the plot shows $\log(1 + f(x))$. Contour lines lie beneath the surface plot.



Rosenbrock's function, log-scaled: two views.

The function $f(x)$ is called the *objective function.* This is the function you wish to minimize. The inequality $x_1^2 + x_2^2 \le 1$ is called a *constraint.* Constraints limit the set of x over which you may search for a minimum. You may have any number of constraints, which may be inequalities or equations.

All Optimization Toolbox optimization functions minimize an objective function. To maximize a function f, apply an optimization routine to minimize $-f$.

Defining the Problem in Toolbox Syntax

To use Optimization Toolbox software, you need to

1. Define your objective function in the MATLAB language, as an M-file or anonymous function. This example will use an M-file.

2 Define your constraint(s) as a separate M-file or anonymous function.

M-file for Objective Function

An M-file is a text file containing MATLAB commands with the extension `.m`. Create a new M-file in any text editor, or use the built-in MATLAB Editor as follows:

1 At the command line type

```
edit rosenbrock
```

The MATLAB Editor opens.

2 In the editor type:

```
function f = rosenbrock(x)
f = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
```

3 Save the file by selecting **File > Save**.

M-File for Constraint Function

Constraint functions must be formulated so that they are in the form $c(x) \le 0$ or $ceq(x) = 0$. The constraint $x_1^2 + x_2^2 \le 1$ needs to be reformulated as $x_1^2 + x_2^2 - 1 \le 0$ in order to have the correct syntax.

Furthermore, toolbox functions that accept nonlinear constraints need to have both equality and inequality constraints defined. In this example there is only an inequality constraint, so you must pass an empty array `[ ]` as the equality constraint function *ceq*.

With these considerations in mind, write a function M-file for the nonlinear constraint:

1 Create a file named `unitdisk.m` containing the following code:

```
function [c, ceq] = unitdisk(x)
c = x(1)^2 + x(2)^2 - 1;
ceq = [ ];
```

2 Save the file `unitdisk.m`.

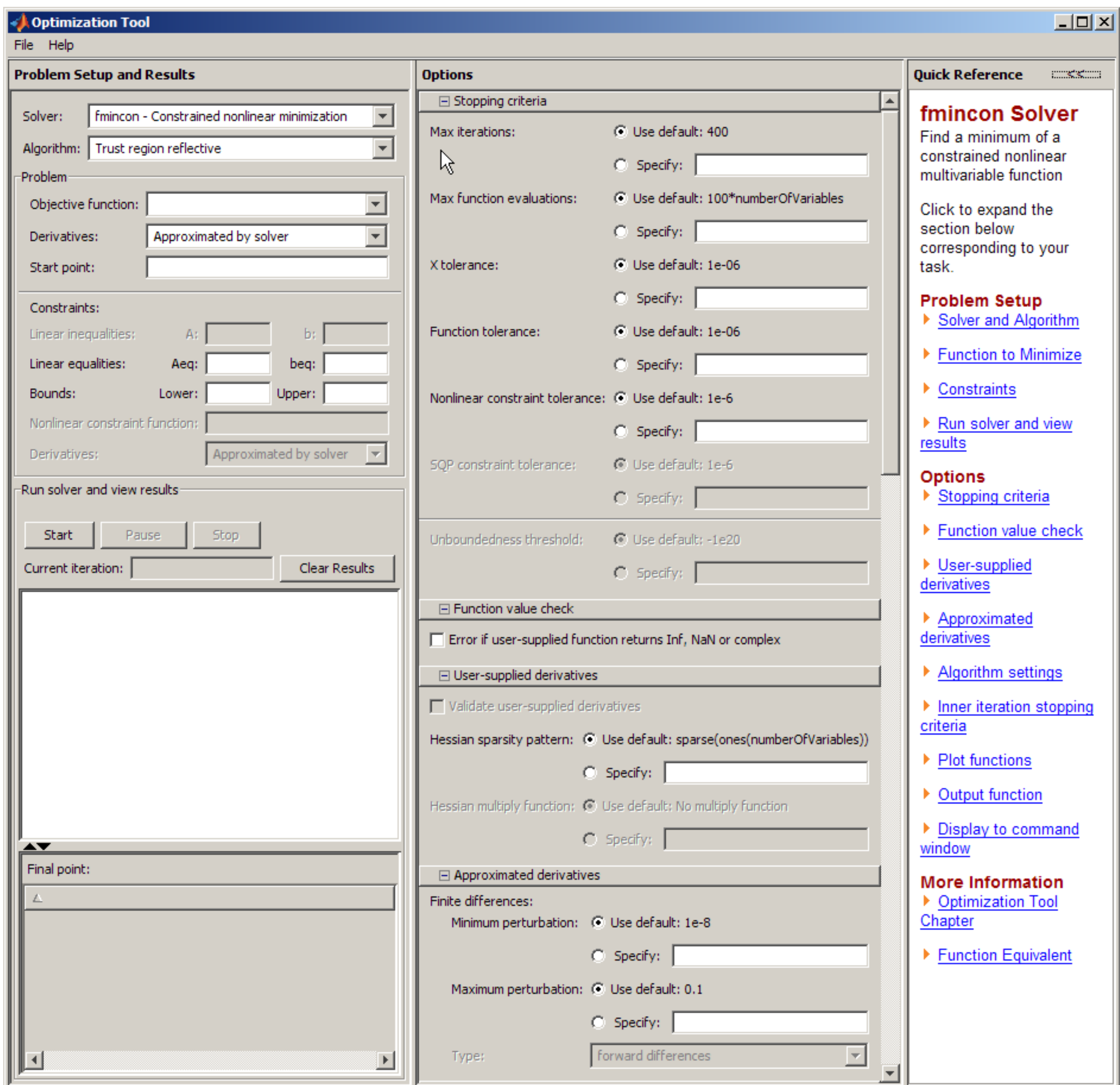
Running the Optimization

There are two ways to run the optimization:

- Using the "Optimization Tool" on page 1-7 Graphical User Interface (GUI)
- Using command line functions; see "Minimizing at the Command Line" on page 1-11.

Optimization Tool

1 Start the Optimization Tool by typing `optimtool` at the command line. The following GUI opens.



For more information about this tool, see Chapter 3, “Optimization Tool”.

2 The default **Solver** `fmincon - Constrained nonlinear minimization` is selected. This solver is appropriate for this problem, since Rosenbrock's function is nonlinear, and the problem has a constraint. For more information about how to choose a solver, see "Choosing a Solver" on page 2-21.

3 In the **Algorithm** pop-up menu choose `Active set`—the default `Trust region reflective` solver doesn't handle nonlinear constraints.

4 For **Objective function** type `@rosenbrock`. The @ character indicates that this is a function handle of the M-file `rosenbrock.m`.

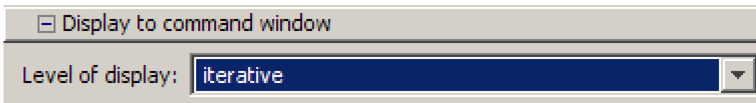
5 For **Start point** type `[0 0]`. This is the initial point where `fmincon` begins its search for a minimum.

6 For **Nonlinear constraint function** type `@unitdisk`, the function handle of `unitdisk.m`.

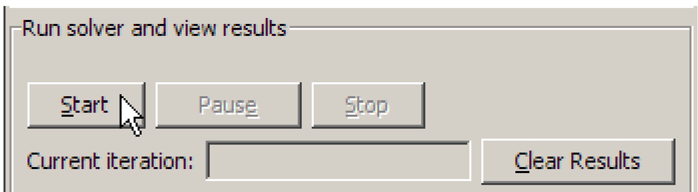
Your **Problem Setup and Results** pane should match this figure.

Solver: fmincon - Constrained nonlinear minimization
Algorithm: Active set
Problem
Objective function: @rosenbrock
Derivatives: Approximated by solver
Start point: [0 0]
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function: @unitdisk
Derivatives: Approximated by solver

7 In the Options pane (center bottom), select `iterative` in the **Level of display** pop-up menu. (If you don't see the option, click ⊞ **Display to command window**.) This shows the progress of `fmincon` in the command window.



8 Click **Start** under **Run solver and view results**.



The following message appears in the box below the **Start** button:

```
Optimization running.
Optimization terminated.
Objective function value: 0.04567571111479972
Optimization terminated: magnitude of directional derivative in search
 direction less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon.
```

Your objective function value may differ slightly, depending on your computer system and version of Optimization Toolbox software.

The message tells you that:

- The search for a constrained optimum ended because the derivative of the objective function is nearly 0 in directions allowed by the constraint.
- The constraint is very nearly satisfied.

The minimizer `x` appears under **Final point**.

Final point:

1	2
0.786	0.618

Minimizing at the Command Line

You can run the same optimization from the command line, as follows.

1 Create an options structure to choose iterative display and the `active-set` algorithm:

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

2 Run the `fmincon` solver with the structure `options`, reporting both the location `x` of the minimizer, and value `fval` attained by the objective function:

```
[x,fval] = fmincon(@rosenbrock,[0 0],...
                    [],[],[],[],[],[],@unitdisk,options)
```

The six sets of empty brackets represent optional constraints that are not being used in this example. See the `fmincon` function reference pages for the syntax.

MATLAB outputs a table of iterations, and the results of the optimization:

```
Optimization terminated: magnitude of directional derivative in search
 direction less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon.
No active inequalities.

x =
    0.7864    0.6177

fval =
    0.0457
```

The message tells you that the search for a constrained optimum ended because the derivative of the objective function is nearly 0 in directions allowed by the constraint, and that the constraint is very nearly satisfied.

Interpreting the Result

The iteration table in the command window shows how MATLAB searched for the minimum value of Rosenbrock's function in the unit disk. This table is the same whether you use Optimization Tool or the command line. MATLAB reports the minimization as follows:

Iter	F-count	f(x)	Max constraint	Line search steplength	Directional derivative	First-order optimality	Procedure
0	3	1	-1				
1	9	0.953127	-0.9375	0.125	9.5	12.5	
2	16	0.808445	-0.8601	0.0625	0.715	12.4	
3	21	0.462347	-0.836	0.25	1.83	5.15	
4	24	0.340677	-0.7969	1	-0.0409	0.811	
5	27	0.300877	-0.7193	1	0.0087	3.72	
6	30	0.261949	-0.6783	1	-0.0348	3.02	
7	33	0.164971	-0.4972	1	-0.0632	2.29	
8	36	0.110766	-0.3427	1	-0.0272	2	
9	40	0.0750932	-0.1592	0.5	-0.00514	2.41	
10	43	0.0580976	-0.007608	1	-0.00908	3.19	
11	47	0.0482475	-0.003783	0.5	-0.0122	1.41	
12	51	0.0464333	-0.001888	0.5	-0.00257	0.726	
13	55	0.0459217	-0.0009431	0.5	-0.000759	0.362	
14	59	0.0457652	-0.0004713	0.5	-0.000247	0.181	
15	63	0.0457117	-0.0002356	0.5	-9.04e-005	0.0906	Hessian modified
16	67	0.0456912	-0.0001178	0.5	-3.69e-005	0.0453	Hessian modified
17	71	0.0456825	-5.889e-005	0.5	-1.64e-005	0.0226	Hessian modified
18	75	0.0456785	-2.944e-005	0.5	-7.67e-006	0.0113	Hessian modified
19	79	0.0456766	-1.472e-005	0.5	-3.71e-006	0.00566	Hessian modified
20	83	0.0456757	-7.361e-006	0.5	-1.82e-006	0.00283	Hessian modified

This table might differ from yours depending on toolbox version and computing platform. The following description applies to the table as displayed.

- The first column, labeled `Iter`, is the iteration number from 0 to 20. `fmincon` took 20 iterations to converge.
- The second column, labeled `F-count`, reports the cumulative number of times Rosenbrock's function was evaluated. The final row shows an `F-count` of 83, indicating that `fmincon` evaluated Rosenbrock's function 83 times in the process of finding a minimum.

- The third column, labeled `f(x)`, displays the value of the objective function. The final value, 0.0456757, is the minimum that is reported in the Optimization Tool **Run solver and view results** box, and at the end of the exit message in the command window.
- The fourth column, `Max constraint`, goes from a value of –1 at the initial value, to very nearly 0, –7.361e–006, at the final iteration. This column shows the value of the constraint function `unitdisk` at each iteration. Since the value of `unitdisk` was nearly 0 at the final iteration, $x_1^2 + x_2^2 \approx 1$ there.

The other columns of the iteration table are described in "Displaying Iterative Output" on page 2-47.

2

Optimization Overview

- "Introduction to Optimization Toolbox Solvers" on page 2-2
- "Writing Objective Functions" on page 2-4
- "Writing Constraints" on page 2-11
- "Passing Extra Parameters" on page 2-17
- "Choosing a Solver" on page 2-21
- "Solver Inputs and Outputs" on page 2-27
- "Default Options Settings" on page 2-41
- "Displaying Iterative Output" on page 2-47
- "Typical Problems and How to Deal with Them" on page 2-54
- "Local vs. Global Optima" on page 2-57
- "Reference" on page 2-62

Introduction to Optimization Toolbox Solvers

There are four general categories of Optimization Toolbox solvers:

- Minimizers

  This group of solvers attempts to find a local minimum of the objective function near a starting point `x0`. They address problems of unconstrained optimization, linear programming, quadratic programming, and general nonlinear programming.

- Multiobjective minimizers

  This group of solvers attempts to either minimize the maximum value of a set of functions (`fminimax`), or to find a location where a collection of functions is below some prespecified values (`fgoalattain`).

- Equation solvers

  This group of solvers attempts to find a solution to a scalar- or vector-valued nonlinear equation $f(x) = 0$ near a starting point `x0`. Equation-solving can be considered a form of optimization because it is equivalent to finding the minimum norm of $f(x)$ near `x0`.

- Least-Squares (curve-fitting) solvers

  This group of solvers attempts to minimize a sum of squares. This type of problem frequently arises in fitting a model to data. The solvers address problems of finding nonnegative solutions, bounded or linearly constrained solutions, and fitting parameterized nonlinear models to data.

For more information see "Problems Handled by Optimization Toolbox Functions" on page 2-21. See "Optimization Decision Table" on page 2-24 for aid in choosing among solvers for minimization.

Minimizers formulate optimization problems in the form

$$\min_x f(x),$$

possibly subject to constraints. $f(x)$ is called an *objective function*. In general, $f(x)$ is a scalar function of type `double`, and x is a vector or scalar of type `double`. However, multiobjective optimization, equation solving, and some sum-of-squares minimizers, can have vector or matrix objective functions $F(x)$

of type `double`. To use Optimization Toolbox solvers for maximization instead of minimization, see “Maximizing an Objective” on page 2-9.

Write the objective function for a solver in the form of an M-file or anonymous function handle. You can supply a gradient $\nabla f(x)$ for many solvers, and you can supply a Hessian for several solvers. See “Writing Objective Functions” on page 2-4. Constraints have a special form, as described in “Writing Constraints” on page 2-11.

Writing Objective Functions

In this section...

"Writing Objective Functions" on page 2-4

"Jacobians of Vector and Matrix Objective Functions" on page 2-6

"Anonymous Function Objectives" on page 2-9

"Maximizing an Objective" on page 2-9

Writing Objective Functions

This section relates to scalar-valued objective functions. For vector-valued or matrix-valued objective functions, see "Jacobians of Vector and Matrix Objective Functions" on page 2-6. For information on how to include extra parameters, see "Passing Extra Parameters" on page 2-17.

An objective function M-file can return one, two, or three outputs. It can return:

- A single double-precision number, representing the value of $f(x)$
- Both $f(x)$ and its gradient $\nabla f(x)$
- All three of $f(x)$, $\nabla f(x)$, and the Hessian matrix $H(x)=\partial^2 f/\partial x_i \partial x_j$

You are not required to provide a gradient for some solvers, and you are never required to provide a Hessian, but providing one or both can lead to faster execution and more reliable answers. If you do not provide a gradient or Hessian, solvers may attempt to estimate them using finite difference approximations or other numerical schemes.

Some solvers do not use gradient or Hessian information. You should "conditionalize" an M-file so that it returns just what is needed:

- $f(x)$ alone
- Both $f(x)$ and $\nabla f(x)$
- All three of $f(x)$, $\nabla f(x)$, and $H(x)$

For example, consider Rosenbrock's function

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

which is described and plotted in "Example: Nonlinear Constrained Minimization" on page 1-4. The gradient of $f(x)$ is

$$\nabla f(x) = \begin{bmatrix} -400\left(x_2 - x_1^2\right)x_1 - 2(1 - x_1) \\ 200\left(x_2 - x_1^2\right) \end{bmatrix},$$

and the Hessian $H(x)$ is

$$H(x) = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}.$$

Function `rosenboth` returns the value of Rosenbrock's function in `f`, the gradient in `g`, and the Hessian in `H` if required:

```
function [f g H] = rosenboth(x)
% Calculate objective f
f = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;

if nargout > 1 % gradient required
    g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
        200*(x(2)-x(1)^2)];

    if nargout > 2 % Hessian required
        H = [1200*x(1)^2-400*x(2)+2, -400*x(1);
            -400*x(1), 200];
    end

end
```

`nargout` checks the number of arguments that a calling function specifies; see "Checking the Number of Input Arguments" in the MATLAB Programming Fundamentals documentation.

The solver fminunc, designed for unconstrained optimization, allows you to minimize Rosenbrock's function. Tell fminunc to use the gradient and Hessian by setting options:

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

Run fminunc starting at [–1, 2]:

```
[x fval] = fminunc(@rosenboth, [-1; 2], options)
Optimization terminated: first-order optimality less than OPTIONS.TolFun,
 and no negative/zero curvature detected in trust region model.
x =
    1.0000
    1.0000
fval =
  1.9310e-017
```

Jacobians of Vector and Matrix Objective Functions

Some solvers, such as fsolve and lsqcurvefit, can have objective functions that are vectors or matrices. The only difference in usage between these types of objective functions and scalar objective functions is the way to write their derivatives. The first-order partial derivatives of a vector-valued or matrix-valued function is called a Jacobian; the first-order partial derivatives of a scalar function is called a gradient.

Jacobians of Vector Functions

If x represents a vector of independent variables, and $F(x)$ is the vector function, the Jacobian $J(x)$ is defined as

$$J_{ij}(x) = \frac{\partial F_i(x)}{\partial x_j}.$$

If F has m components, and x has k components, J is a m-by-k matrix.

For example, if

$$F(x) = \begin{bmatrix} x_1^2 + x_2 x_3 \\ \sin(x_1 + 2x_2 - 3x_3) \end{bmatrix},$$

then $J(x)$ is

$$J(x) = \begin{bmatrix} 2x_1 & x_3 & x_2 \\ \cos(x_1 + 2x_2 - 3x_3) & 2\cos(x_1 + 2x_2 - 3x_3) & -3\cos(x_1 + 2x_2 - 3x_3) \end{bmatrix}.$$

Jacobians of Matrix Functions

The Jacobian of a matrix $F(x)$ is defined by changing the matrix to a vector, column by column. For example, the matrix

$$F = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \\ F_{31} & F_{32} \end{bmatrix}$$

is rewritten as a vector f:

$$f = \begin{bmatrix} F_{11} \\ F_{21} \\ F_{31} \\ F_{12} \\ F_{22} \\ F_{32} \end{bmatrix}.$$

The Jacobian of F is defined as the Jacobian of f,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

If F is an m-by-n matrix, and x is a k-vector, the Jacobian is a mn-by-k matrix.

For example, if

$$F(x) = \begin{bmatrix} x_1 x_2 & x_1^3 + 3x_2^2 \\ 5x_2 - x_1^4 & x_2 / x_1 \\ 4 - x_2^2 & x_1^3 - x_2^4 \end{bmatrix},$$

the Jacobian of F is

$$J(x) = \begin{bmatrix} x_2 & x_1 \\ -4x_1^3 & 5 \\ 0 & -2x_2 \\ 3x_1^2 & 6x_2 \\ -x_2 / x_1^2 & 1/x_1 \\ 3x_1^2 & -4x_2^3 \end{bmatrix}.$$

Jacobians with Matrix-Valued Independent Variables

If x is a matrix, the Jacobian of $F(x)$ is defined by changing the matrix x to a vector, column by column. For example, if

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix},$$

then the gradient is defined in terms of the vector

$$x = \begin{bmatrix} x_{11} \\ x_{21} \\ x_{12} \\ x_{22} \end{bmatrix}.$$

With

$$F = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \\ F_{31} & F_{32} \end{bmatrix},$$

and f the vector form of F as above, the Jacobian of $F(X)$ is defined to be the Jacobian of $f(x)$:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

So, for example,

$$J(3,2)=\frac{\partial f(3)}{\partial x(2)}=\frac{\partial F_{31}}{\partial X_{21}},\ \text{and}\ J(5,4)=\frac{\partial f(5)}{\partial x(4)}=\frac{\partial F_{22}}{\partial X_{22}}.$$

If F is an m-by-n matrix, and x is a j-by-k matrix, the Jacobian is a mn-by-jk matrix.

Anonymous Function Objectives

Anonymous functions are useful for writing simple objective functions, without gradient or Hessian information. Rosenbrock's function is simple enough to write as an anonymous function:

```
anonrosen = @(x)(100*(x(2) - x(1)^2)^2 + (1 - x(1))^2);
```

Check that this evaluates correctly at (–1,2):

```
>> anonrosen([-1 2])
ans =
   104
```

Using `anonrosen` in `fminunc` yields the following results:

```
[x fval] = fminunc(anonrosen, [-1; 2])
Warning: Gradient must be provided for trust-region method;
  using line-search method instead.
> In fminunc at 265
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
x =
    1.0000
    1.0000
fval =
  1.2262e-010
```

Maximizing an Objective

All solvers are designed to minimize an objective function. If you have a maximization problem, that is, a problem of the form

$$\max_x f(x),$$

then define $g(x) = -f(x)$, and minimize g.

For example, to find the maximum of tan(cos(x)) near $x = 5$, evaluate:

```
[x fval] = fminunc(@(x)-tan(cos(x)),5)
Warning: Gradient must be provided for trust-region method;
  using line-search method instead.
> In fminunc at 265
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
x =
    6.2832
fval =
   -1.5574
```

The maximum is 1.5574 (the negative of the reported `fval`), and occurs at $x = 6.2832$. This is correct since, to 5 digits, the maximum is tan(1) = 1.5574, which occurs at $x = 2\pi = 6.2832$.

Writing Constraints

In this section...

"Types of Constraints" on page 2-11
"Bound Constraints" on page 2-12
"Linear Inequality Constraints" on page 2-13
"Linear Equality Constraints" on page 2-13
"Nonlinear Constraints" on page 2-14
"An Example Using All Types of Constraints" on page 2-15

Types of Constraints

Optimization Toolbox solvers have special forms for constraints. Constraints are separated into the following types:

- "Bound Constraints" on page 2-12 — Lower and upper bounds on individual components: $x \geq l$ and $x \leq u$.
- "Linear Inequality Constraints" on page 2-13 — $A \cdot x \leq b$. A is an m-by-n matrix, which represents m constraints for an n-dimensional vector x. b is m-dimensional.
- "Linear Equality Constraints" on page 2-13 — $Aeq \cdot x = beq$. This is a system of equations.
- "Nonlinear Constraints" on page 2-14 — $c(x) \leq 0$ and $ceq(x) = 0$. Both c and ceq are scalars or vectors representing several constraints.

Optimization Toolbox functions assume that inequality constraints are of the form $c_i(x) \leq 0$ or $A\ x \leq b$. Greater-than constraints are expressed as less-than constraints by multiplying them by –1. For example, a constraint of the form $c_i(x) \geq 0$ is equivalent to the constraint $-c_i(x) \leq 0$. A constraint of the form $A \cdot x \geq b$ is equivalent to the constraint $-A \cdot x \leq -b$. For more information, see "Linear Inequality Constraints" on page 2-13 and "Nonlinear Constraints" on page 2-14.

You can sometimes write constraints in a variety of ways. To make the best use of the solvers, use the lowest numbered constraints possible:

1. Bounds
2. Linear equalities
3. Linear inequalities
4. Nonlinear equalities
5. Nonlinear inequalities

For example, with a constraint $5x \leq 20$, use a bound $x \leq 4$ instead of a linear inequality or nonlinear inequality.

For information on how to pass extra parameters to constraint functions, see "Passing Extra Parameters" on page 2-17.

Bound Constraints

Lower and upper bounds on the components of the vector x. You need not give gradients for this type of constraint; solvers calculate them automatically. Bounds do not affect Hessians.

If you know bounds on the location of your optimum, then you may obtain faster and more reliable solutions by explicitly including these bounds in your problem formulation.

Bounds are given as vectors, with the same length as x.

- If a particular component is not bounded below, use `–Inf` as the bound; similarly, use `Inf` if a component is not bounded above.
- If you have only bounds of one type (upper or lower), you do not need to write the other type. For example, if you have no upper bounds, you do not need to supply a vector of `Inf`s. Also, if only the first m out of n components are bounded, then you need only supply a vector of length m containing bounds.

For example, suppose your bounds are:

- $x_3 \geq 8$
- $x_2 \leq 3$

Write the constraint vectors as

- l = [–Inf; –Inf; 8]
- u = [Inf; 3] or u = [Inf; 3; Inf]

Linear Inequality Constraints

Linear inequality constraints are of the form $A\cdot x \le b$. When A is m-by-n, this represents m constraints on a variable x with n components. You supply the m-by-n matrix A and the m-component vector b. You do not need to give gradients for this type of constraint; solvers calculate them automatically. Linear inequalities do not affect Hessians.

For example, suppose that you have the following linear inequalities as constraints:

$$
\begin{aligned}
&x_1 + x_3 \le 4,\\
&2x_2 - x_3 \ge -2,\\
&x_1 - x_2 + x_3 - x_4 \ge 9.
\end{aligned}
$$

Here $m = 3$ and $n = 4$.

Write these using the following matrix A and vector b:

$$
A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & -2 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix},
$$

$$
b = \begin{bmatrix} 4 \\ 2 \\ -9 \end{bmatrix}.
$$

Notice that the “greater than” inequalities were first multiplied by –1 in order to get them into “less than” inequality form.

Linear Equality Constraints

Linear equalities are of the form $Aeq\cdot x = beq$. This represents m equations with n-component vector x. You supply the m-by-n matrix Aeq and the m-component vector beq. You do not need to give a gradient for this type of

constraint; solvers calculate them automatically. Linear equalities do not affect Hessians. The form of this type of constraint is exactly the same as for "Linear Inequality Constraints" on page 2-13. Equalities rather than inequalities are implied by the position in the input argument list of the various solvers.

Nonlinear Constraints

Nonlinear inequality constraints are of the form $c(x) \le 0$, where c is a vector of constraints, one component for each constraint. Similarly, nonlinear equality constraints are of the form $ceq(x) = 0$. If you provide gradients for c and ceq, your solver may run faster and give more reliable results.

For example, suppose that you have the following inequalities as constraints:

$$\frac{x_1^2}{9} + \frac{x_2^2}{4} \le 1,$$

$$x_2 \ge x_1^2 - 1.$$

Write these constraints in an M-file as follows:

```
function [c,ceq]=ellipseparabola(x)
% Inside the ellipse bounded by (-3<x<3),(-2<y<2)
% Above the line y=x^2-1
c(1) = (x(1)^2)/9 + (x(2)^2)/4 - 1;
c(2) = x(1)^2 - x(2) - 1;
ceq = [];
end
```

The constraint function returns empty [ ] as the nonlinear equality function. Nonlinear constraint functions must return both inequality and equality constraints, even if they do not both exist. Also, both inequalities were put into ≤ 0 form.

To include gradient information, write a conditionalized function as follows:

```
function [c,ceq,gradc,gradceq]=ellipseparabola(x)
% Inside the ellipse bounded by (-3<x<3),(-2<y<2)
% Above the line y=x^2-1
c(1) = x(1)^2/9 + x(2)^2/4 - 1;
```

```
c(2) = x(1)^2 - x(2) - 1;
ceq = [];

if nargout > 2
    gradc = [2*x(1)/9, 2*x(1);...
              x(2)/2, -1];
    gradceq = [];
end
```

See “Writing Objective Functions” on page 2-4 for information on conditionalized gradients. The gradient matrix is of the form

$\texttt{gradc}_{i,j} = [\partial \texttt{c}(j)/\partial x_i]$.

The first column of the gradient matrix is associated with `c(1)`, and the second column is associated with `c(2)`. This is the transpose of the form of Jacobians.

To have a solver use gradients of nonlinear constraints, you must indicate that you have supplied them by using `optimset`:

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

Make sure to pass the options structure to your solver:

```
[x,fval] = fmincon(@myobj,x0,A,b,Aeq,beq,lb,ub,...
           @ellipseparabola,options)
```

An Example Using All Types of Constraints

This section contains an example of a nonlinear minimization problem with all possible types of constraints. The objective function is in the subfunction `myobj(x)`. The nonlinear constraints are in the subfunction `myconstr(x)`. Gradients are not used in this example.

```
function fullexample
x0 = [1; 4; 5; 2; 5];
lb = [-Inf; -Inf;  0; -Inf;   1];
ub = [ Inf;  Inf; 20];
Aeq = [1 -0.3 0 0 0];
beq = 0;
A = [0 0  0 -1  0.1
     0 0  0  1 -0.5
```

```
     0 0 -1  0  0.9];
b = [0; 0; 0];

[x,fval,exitflag]=fmincon(@myobj,x0,A,b,Aeq,beq,lb,ub,...
                                        @myconstr)

%---------------------------------------------------------
function f = myobj(x)

f = 6*x(2)*x(5) + 7*x(1)*x(3) + 3*x(2)^2;

%---------------------------------------------------------
function [c, ceq] = myconstr(x)

c = [x(1) - 0.2*x(2)*x(5) - 71
     0.9*x(3) - x(4)^2 - 67];
ceq = 3*x(2)^2*x(5) + 3*x(1)^2*x(3) - 20.875;
```

Calling `fullexample` produces the following display in the command window:

```
fullexample
Warning: Trust-region-reflective method does not currently solve this type of problem,
 using active-set (line search) instead.
> In fmincon at 317
  In fullexample at 12
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
                          3
x =
    0.6114
    2.0380
    1.3948
    0.3585
    1.5498
fval =
   37.3806
exitflag =
     1
```

Passing Extra Parameters

Sometimes objective or constraint functions have parameters in addition to the independent variable. There are three methods of including these parameters:

- "Anonymous Functions" on page 2-17
- "Nested Functions" on page 2-19
- "Global Variables" on page 2-20

Global variables are troublesome because they do not allow names to be reused among functions. It is better to use one of the other two methods.

For example, suppose you want to minimize the function

$$f(x) = \left(a - bx_1^2 + x_1^{4/3}\right)x_1^2 + x_1x_2 + \left(-c + cx_2^2\right)x_2^2, \tag{2-1}$$

for different values of a, b, and c. Solvers accept objective functions that depend only on a single variable (x in this case). The following sections show how to provide the additional parameters a, b, and c. The solutions are for parameter values $a = 4$, $b = 2.1$, and $c = 4$ near $x_0 = [0.5\ 0.5]$ using `fminunc`.

Anonymous Functions

To pass parameters using anonymous functions:

1 Write an M-file containing the following code:

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

2 Assign values to the parameters and define a function handle `f` to an anonymous function by entering the following commands at the MATLAB prompt:

```
a = 4; b = 2.1; c = 4; % Assign parameter values
x0 = [0.5,0.5];
```

```
f = @(x)parameterfun(x,a,b,c)
```

3 Call the solver fminunc with the anonymous function:

```
[x,fval] = fminunc(f,x0)
```

The following output is displayed in the command window:

```
Warning: Gradient must be provided for trust-region method;
  using line-search method instead.
> In fminunc at 265
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
x =
   -0.0898    0.7127
fval =
   -1.0316
```

Note The parameters passed in the anonymous function are those that exist at the time the anonymous function is created. Consider the example

```
a = 4; b = 2.1; c = 4;
f = @(x)parameterfun(x,a,b,c)
```

Suppose you subsequently change, a to 3 and run

```
[x,fval] = fminunc(f,x0)
```

You get the same answer as before, since parameterfun uses a = 4, the value when parameterfun was created.

To change the parameters that are passed to the function, renew the anonymous function by reentering it:

```
a = 3;
f = @(x)parameterfun(x,a,b,c)
```

You can create anonymous functions of more than one argument. For example, to use lsqcurvefit, first create 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
```

Now call `lsqcurvefit`:

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

Nested Functions

To pass the parameters for Equation 2-1 via a nested function, write a single M-file that

- Accepts `a`, `b`, `c`, and `x0` as inputs
- Contains the objective function as a nested function
- Calls `fminunc`

Here is the code for the M-file for this example:

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

Note that the objective function is computed in the nested function `nestedfun`, which has access to the variables `a`, `b`, and `c`.

To run the optimization, enter:

```
a = 4; b = 2.1; c = 4;% Assign parameter values
```

```
x0 = [0.5,0.5];
[x,fval] = runnested(a,b,c,x0)
```

The output is the same as in “Anonymous Functions” on page 2-17.

Global Variables

Global variables can be troublesome, it is better to avoid using them. To use global variables, declare the variables to be global in the workspace and in the functions that use the variables.

1. Write an M-file containing code for your function:

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

2. In your MATLAB workspace, define the variables and run `fminunc`:

   ```
   global a b c;
   a = 4; b = 2.1; c = 4; % Assign parameter values
   x0 = [0.5,0.5];
   [x,fval] = fminunc(@globalfun,x0)
   ```

The output is the same as in “Anonymous Functions” on page 2-17.

Choosing a Solver

In this section...

"Problems Handled by Optimization Toolbox Functions" on page 2-21

"Optimization Decision Table" on page 2-24

Problems Handled by Optimization Toolbox Functions

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

Minimization Problems

Type	Formulation	Solver
Scalar minimization	$\min_x f(x)$ such that $l < x < u$ (x is scalar)	`fminbnd`
Unconstrained minimization	$\min_x f(x)$	`fminunc`, `fminsearch`
Linear programming	$\min_x f^T x$ such that $A \cdot x \leq b, \quad Aeq \cdot x = beq, \quad l \leq x \leq u$	`linprog`
Quadratic programming	$\min_x \frac{1}{2} x^T Hx + c^T x$ such that $A \cdot x \leq b, \quad Aeq \cdot x = beq, \quad l \leq x \leq u$	`quadprog`

Minimization Problems (Continued)

Type	Formulation	Solver
Constrained minimization	$\min_x f(x)$ such that $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$	`fmincon`
Semi-infinite minimization	$\min_x f(x)$ such that $K(x,w) \le 0$ for all w, $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$	`fseminf`
Binary integer programming	$\min_x f^T x$ such that $A \cdot x \le b$, $Aeq \cdot x = beq$, x binary	`bintprog`

Multiobjective Problems

Type	Formulation	Solver
Goal attainment	$\min_{x,\gamma} \gamma$ such that $F(x) - w \cdot \gamma \le \text{goal}$, $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$	`fgoalattain`
Minimax	$\min_x \max_i F_i(x)$ such that $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$	`fminimax`

Equation Solving Problems

Type	Formulation	Solver
Linear equations	$C \cdot x = d$, n equations, n variables	\ (matrix left division)
Nonlinear equation of one variable	$f(x) = 0$	`fzero`
Nonlinear equations	$F(x) = 0$, n equations, n variables	`fsolve`

Least-Squares (Model-Fitting) Problems

Type	Formulation	Solver
Linear least-squares	$\min_{x} \lVert C \cdot x - d \rVert_2^2$ m equations, n variables	\ (matrix left division)
Nonnegative linear-least-squares	$\min_{x} \lVert C \cdot x - d \rVert_2^2$ such that $x \geq 0$	`lsqnonneg`
Constrained linear-least-squares	$\min_{x} \lVert C \cdot x - d \rVert_2^2$ such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`lsqlin`
Nonlinear least-squares	$\min_{x} \lVert F(x) \rVert_2^2 = \min_{x} \sum_i F_i^2(x)$ such that $lb \leq x \leq ub$	`lsqnonlin`
Nonlinear curve fitting	$\min_{x} \lVert F(x, xdata) - ydata \rVert_2^2$ such that $lb \leq x \leq ub$	`lsqcurvefit`

Optimization Decision Table

The following table is designed to help you choose a solver. It does not address multiobjective optimization or equation solving. There are more details on all the solvers in "Problems Handled by Optimization Toolbox Functions" on page 2-21.

Use the table as follows:

1. Identify your objective function as one of five types:
   - Linear
   - Quadratic
   - Sum-of-squares (Least squares)
   - Smooth nonlinear
   - Nonsmooth
2. Identify your constraints as one of five types:
   - None (unconstrained)
   - Bound
   - Linear (including bound)
   - General smooth
   - Discrete (integer)
3. Use the table to identify a relevant solver.

In this table:

- Blank entries means there is no Optimization Toolbox solver specifically designed for this type of problem.
- * means relevant solvers are found in Genetic Algorithm and Direct Search Toolbox functions (licensed separately from Optimization Toolbox solvers).
- `fmincon` applies to most smooth objective functions with smooth constraints. It is not listed as a preferred solver for least squares or linear or quadratic programming because the listed solvers are usually more efficient.

- The table has suggested functions, but it is not meant to unduly restrict your choices. For example, `fmincon` is known to be effective on some non-smooth problems.
- The Genetic Algorithm and Direct Search Toolbox function `ga` can be programmed to address discrete problems. It is not listed in the table because additional programming is needed to solve discrete problems.

Solvers by Objective and Constraint

Constraint Type	Objective Type				
	Linear	**Quadratic**	**Least Squares**	**Smooth nonlinear**	**Nonsmooth**
None	n/a (f = const, or min = $-\infty$)	`quadprog`, Theory, Examples	`\`, `lsqcurvefit`, `lsqnonlin`, Theory, Examples	`fminsearch`, `fminunc`, Theory, Examples	`fminsearch`, *
Bound	`linprog`, Theory, Examples	`quadprog`, Theory, Examples	`lsqcurvefit`, `lsqlin`, `lsqnonlin`, `lsqnonneg`, Theory, Examples	`fminbnd`, `fmincon`, `fseminf`, Theory, Examples	*
Linear	`linprog`, Theory, Examples	`quadprog`, Theory, Examples	`lsqlin`, Theory, Examples	`fmincon`, `fseminf`, Theory, Examples	*
General smooth	`fmincon`, Theory, Examples	`fmincon`, Theory, Examples	`fmincon`, Theory, Examples	`fmincon`, `fseminf`, Theory, Examples	*
Discrete	`bintprog`, Theory, Example				

Note This table does not list multiobjective solvers nor equation solvers. See "Problems Handled by Optimization Toolbox Functions" on page 2-21 for a complete list of problems addressed by Optimization Toolbox functions.

Solver Inputs and Outputs

In this section...

"Iterations and Function Counts" on page 2-27

"First-Order Optimality Measure" on page 2-28

"Tolerances and Stopping Criteria" on page 2-31

"Lagrange Multiplier Structures" on page 2-32

"Output Structures" on page 2-33

"Output Functions" on page 2-33

Iterations and Function Counts

In general, Optimization Toolbox solvers iterate to find an optimum. This means a solver begins at an initial value x_0, performs some intermediate calculations that eventually lead to a new point x_1, and then repeats the process to find successive approximations x_2, x_3, ... of the local minimum. Processing stops after some number of iterations k.

At any step, intermediate calculations may involve evaluating the objective function and constraints, if any, at points near the current iterate x_i. For example, the solver may estimate a gradient by finite differences. At each of these nearby points, the function count (`F-count`) is increased by one.

- If there are no constraints, the `F-count` reports the total number of objective function evaluations.
- If there are constraints, the `F-count` reports only the number of points where function evaluations took place, not the total number of evaluations of constraint functions.
- If there are many constraints, the `F-count` can be significantly less than the total number of function evaluations.

`F-count` is a header in the iterative display for many solvers. For an example, see "Interpreting the Result" on page 1-12.

The `F-count` appears in the output structure as `output.funcCount`. This enables you to access the evaluation count programmatically. For more information on output structures, see “Output Structures” on page 2-33.

Note Intermediate calculations do not count towards the reported number of iterations k. The number of iterations is the total number of steps x_i, $1 \le i \le k$.

First-Order Optimality Measure

First-order optimality is a measure of how close a point x is to optimal. It is used in all smooth solvers, constrained and unconstrained, though it has different meanings depending on the problem and solver. For more information about first-order optimality, see [1].

The tolerance `TolFun` relates to the first-order optimality measure. If the optimality measure is less than `TolFun`, the solver iterations will end.

Unconstrained Optimality

For a smooth unconstrained problem,

$$\min_x f(x),$$

the optimality measure is the infinity-norm (i.e., maximum absolute value) of $\nabla f(x)$:

$$\text{First-order optimality measure} = \max_i \left| (\nabla f(x))_i \right| = \|\nabla f(x)\|_\infty .$$

This measure of optimality is based on the familiar condition for a smooth function to achieve a minimum: its gradient must be zero. For unconstrained problems, when the first-order optimality measure is nearly zero, the objective function has gradient nearly zero, so the objective function could be nearly minimized. If the first-order optimality measure is not small, the objective function is not minimized.

Constrained Optimality—Theory

The theory behind the definition of first-order optimality measure for constrained problems. The definition as used in Optimization Toolbox functions is in “Constrained Optimality in Solver Form” on page 2-30.

For a smooth constrained problem let g and h be vector functions representing all inequality and equality constraints respectively (i.e., bound, linear, and nonlinear constraints):

$$\min_x f(x) \text{ subject to } g(x) \le 0,\ h(x) = 0.$$

The meaning of first-order optimality in this case is more involved than for unconstrained problems. The definition is based on the Karush-Kuhn-Tucker (KKT) conditions. The KKT conditions are analogous to the condition that the gradient must be zero at a minimum, modified to take constraints into account. The difference is that the KKT conditions hold for constrained problems.

The KKT conditions are given via an auxiliary Lagrangian function

$$L(x,\lambda) = f(x) + \sum \lambda_{g,i} g_i(x) + \sum \lambda_{h,i} h_i(x). \tag{2-2}$$

The vector λ, which is the concatenation of λ_g and λ_h, is called the Lagrange multiplier vector. Its length is the total number of constraints.

The KKT conditions are:

$$\nabla_x L(x,\lambda) = 0, \tag{2-3}$$

$$\lambda_{g,i} g_i(x) = 0 \ \forall i, \tag{2-4}$$

$$\begin{cases} g(x) \le 0, \\ h(x) = 0, \\ \lambda_{g,i} \ge 0. \end{cases} \tag{2-5}$$

The three expressions in Equation 2-5 are not used in the calculation of optimality measure.

The optimality measure associated with Equation 2-3 is

$$\left\|\nabla_x L(x,\lambda\right\| = \left\|\nabla f(x) + \sum \lambda_{g,i} \nabla g_i(x) + \sum \lambda_{h,i} \nabla h_{h,i}(x)\right\|. \tag{2-6}$$

The optimality measure associated with Equation 2-4 is

$$\left\|\overrightarrow{\lambda_g g(x)}\right\|, \tag{2-7}$$

where the infinity norm (maximum) is used for the vector $\overrightarrow{\lambda_{g,i} g_i}(x)$.

The combined optimality measure is the maximum of the values calculated in Equation 2-6 and Equation 2-7. In solvers that accept nonlinear constraint functions, constraint violations $g(x) > 0$ or $|h(x)| > 0$ are measured and reported as tolerance violations; see "Tolerances and Stopping Criteria" on page 2-31.

Constrained Optimality in Solver Form

The first-order optimality measure used by toolbox solvers is expressed as follows for constraints given separately by bounds, linear functions, and nonlinear functions. The measure is the maximum of the following two norms, which correspond to Equation 2-6 and Equation 2-7:

$$\begin{aligned}\left\|\nabla_x L(x,\lambda\right\| = \Big\|\nabla f(x) &+ A^T \lambda_{ineqlin} + Aeq^T \lambda_{eqlin} \\ &+ \sum \lambda_{ineqnonlin,i} \nabla c_i(x) + \sum \lambda_{eqnonlin,i} \nabla ceq_i(x)\Big\|,\end{aligned} \tag{2-8}$$

$$\left\|\overrightarrow{|l_i - x_i| \lambda_{lower,i}}, \overrightarrow{|x_i - u_i| \lambda_{upper,i}}, \overrightarrow{|(Ax-b)_i| \lambda_{ineqlin,i}}, \overrightarrow{|c_i(x)| \lambda_{ineqnonlin,i}}\right\|, \tag{2-9}$$

where the infinity norm (maximum) is used for the vector in Equation 2-8 and in Equation 2-9. The subscripts on the Lagrange multipliers correspond to solver Lagrange multiplier structures; see "Lagrange Multiplier Structures" on page 2-32. The summations in Equation 2-8 range over all constraints. If a bound is ±`Inf`, that term is not considered constrained, so is not part of the summation.

For some large-scale problems with only linear equalities, the first-order optimality measure is the infinity norm of the *projected* gradient (i.e., the gradient projected onto the nullspace of `Aeq`).

Tolerances and Stopping Criteria

The number of iterations in an optimization depends on a solver's *stopping criteria*. These criteria include:

- First-order optimality measure
- Tolerance `TolX`
- Tolerance `TolFun`
- Tolerance `TolCon`
- Bound on number of iterations taken `MaxIter`
- Bound on number of function evaluations `MaxFunEvals`

First-order optimality measure is defined in "First-Order Optimality Measure" on page 2-28. Iterations and function evaluations are discussed in "Iterations and Function Counts" on page 2-27. The remainder of this section describes how Optimization Toolbox solvers use stopping criteria to terminate optimizations.

- `TolX` is a lower bound on the size of a step, meaning the norm of $(x_i - x_{i+1})$. If the solver attempts to take a step that is smaller than `TolX`, the iterations end. `TolX` is sometimes used as a *relative* bound, meaning iterations end when $|(x_i - x_{i+1})| <$ `TolX`$*(1 + |x_i|)$, or a similar relative measure.
- `TolFun` is a lower bound on the change in the value of the objective function during a step. If $|f(x_i) - f(x_{i+1})| <$ `TolFun`, the iterations end. `TolFun` is sometimes used as a *relative* bound, meaning iterations end when $|f(x_i) - f(x_{i+1})| <$ `TolFun`$(1 + |f(x_i)|)$, or a similar relative measure.
- `TolFun` is also a bound on the first-order optimality measure. If the optimality measure is less than `TolFun`, the iterations end. `TolFun` can also be a relative bound.
- `TolCon` is an upper bound on the magnitude of any constraint functions. If a solver returns a point x with $c(x) >$ `TolCon` or $|ceq(x)| >$ `TolCon`, the

solver reports that the constraints are violated at x. `TolCon` can also be a relative bound.

Note `TolCon` operates differently from other tolerances. If `TolCon` is not satisfied (i.e., if the magnitude of the constraint function exceeds `TolCon`), the solver attempts to continue, unless it is halted for another reason. A solver does not halt simply because `TolCon` is satisfied.

There are two other tolerances that apply to particular solvers: `TolPCG` and `MaxPCGIter`. These relate to preconditioned conjugate gradient steps. For more information, see "Preconditioned Conjugate Gradient Method" on page 4-23.

There are several tolerances that apply only to the interior-point algorithm in the solver `fmincon`. See "Optimization Options" on page 7-7 for more information.

Lagrange Multiplier Structures

Constrained optimization involves a set of Lagrange multipliers, as described in "First-Order Optimality Measure" on page 2-28. Solvers return estimated Lagrange multipliers in a structure. The structure is called `lambda`, since the conventional symbol for Lagrange multipliers is the Greek letter lambda (λ). The structure separates the multipliers into the following types, called fields:

- `lower`, associated with lower bounds
- `upper`, associated with upper bounds
- `eqlin`, associated with linear equalities
- `ineqlin`, associated with linear inequalities
- `eqnonlin`, associated with nonlinear equalities
- `ineqnonlin`, associated with nonlinear inequalities

To access, for example, the nonlinear inequality field of a Lagrange multiplier structure, enter `lambda.inqnonlin`. To access the third element of the Lagrange multiplier associated with lower bounds, enter `lambda.lower(3)`.

The content of the Lagrange multiplier structure depends on the solver. For example, linear programming has no nonlinearities, so it does not have `eqnonlin` or `ineqnonlin` fields. Each applicable solver's function reference pages contains a description of its Lagrange multiplier structure under the heading "Outputs."

Output Structures

An *output structure* contains information on a solver's result. All solvers can return an output structure. To obtain an output structure, invoke the solver with the output structure in the calling syntax. For example, to get an output structure from `lsqnonlin`, use the syntax

```
[x,resnorm,residual,exitflag,output] = lsqnonlin(...)
```

You can also obtain an output structure by running a problem using the Optimization Tool. All results exported from Optimization Tool contain an output structure.

The contents of the output structure are listed in each solver's reference pages. For example, the output structure returned by `lsqnonlin` contains `firstorderopt`, `iterations`, `funcCount`, `cgiterations`, `stepsize`, `algorithm`, and `message`. To access, for example, the message, enter `output.message`.

Optimization Tool exports results in a structure. The results structure contains the output structure. To access, for example, the number of iterations, use the syntax `optimresults.output.iterations`.

You can also see the contents of an output structure by double-clicking the output structure in the MATLAB Workspace pane.

Output Functions

Introduction

For some problems, you might want 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 do this, create an

output function that the optimization function calls at each iteration. See "Output Function" on page 7-17 for details and syntax.

Generally, the solvers that can employ an output function are the ones that can take nonlinear functions as inputs. You can determine which solvers can have an output function by looking in the Options section of function reference pages, or by checking whether the **Output function** option is available in the Optimization Tool GUI for a solver.

Example: Using Output Functions

- "What the Example Contains" on page 2-34
- "Writing the Output Function" on page 2-35
- "Writing the Example M-File" on page 2-36
- "Running the Example" on page 2-37

What the Example Contains. The following example continues the one in "Example: Nonlinear Inequality Constraints" on page 4-44, which calls the function `fmincon` at the command line to solve a nonlinear, constrained optimization problem. The example in this section uses an M-file to call `fmincon`. The M-file also contains all the functions needed for the example, including:

- The objective function
- The constraint function
- An output function that records the history of points computed by the algorithm for `fmincon`. At each iteration of the algorithm for `fmincon`, the output function:
  - Plots the current point computed by the algorithm.
  - Stores the point and its corresponding objective function value in a variable called `history`, and stores the current search direction in a variable called `searchdir`. The search direction is a vector that points in the direction from the current point to the next one.

The code for the M-file is here: "Writing the Example M-File" on page 2-36.

Writing the Output Function. You specify the output function in the `options` structure

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

where `outfun` is the name of the output function. When you call an optimization function with `options` as an input, the optimization function calls `outfun` at each iteration of its algorithm.

In general, `outfun` can be any MATLAB function, but in this example, it is a nested subfunction of the M-file described in "Writing the Example M-File" on page 2-36. The following code defines the output function:

```
function stop = outfun(x,optimValues,state)
stop = false;

   switch state
       case 'init'
           hold on
       case 'iter'
           % Concatenate current point and objective function
           % value with history. x must be a row vector.
           history.fval = [history.fval; optimValues.fval];
           history.x = [history.x; x];
           % 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'
           hold off
       otherwise
   end
end
```

See "Using Function Handles with Nested Functions" in the MATLAB Programming Fundamentals documentation for more information about nested functions.

The arguments that the optimization function passes to `outfun` are:

- `x` — The point computed by the algorithm at the current iteration
- `optimValues` — Structure containing data from the current iteration

  The example uses the following fields of `optimValues`:

  - `optimValues.iteration` — Number of the current iteration
  - `optimValues.fval` — Current objective function value
  - `optimValues.searchdirection` — Current search direction
- `state` — The current state of the algorithm (`'init'`, `'interrupt'`, `'iter'`, or `'done'`)

For more information about these arguments, see "Output Function" on page 7-17.

Writing the Example M-File. To create the M-file for this example:

1. Open a new M-file in the MATLAB Editor.
2. Copy and paste the following code into the M-file:

```
function [history,searchdir] = runfmincon

% Set up shared variables with OUTFUN
history.x = [];
history.fval = [];
searchdir = [];

% call optimization
x0 = [-1 1];
options = optimset('outputfcn',@outfun,'display','iter',...
'Algorithm','active-set');
xsol = fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options);

 function stop = outfun(x,optimValues,state)
     stop = false;

     switch state
```

```
            case 'init'
                hold on
            case 'iter'
            % Concatenate current point and objective function
            % value with history. x must be a row vector.
              history.fval = [history.fval; optimValues.fval];
              history.x = [history.x; x];
            % Concatenate current search direction with
            % searchdir.
              searchdir = [searchdir;...
                           optimValues.searchdirection'];
              plot(x(1),x(2),'o');
            % Label points with iteration number and add title.
              text(x(1)+.15,x(2),...
                   num2str(optimValues.iteration));
              title('Sequence of Points Computed by fmincon');
            case 'done'
                hold off
            otherwise
        end
    end

    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);
    end

    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 = [];
    end
   end
```

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

Running the Example. To run the example, enter:

```
[history searchdir] = runfmincon;
```

This displays the following iterative output in the Command Window.

```
                                Max     Line search  Directional  First-order
Iter F-count        f(x)   constraint   steplength   derivative   optimality Procedure
    0      3      1.8394          0.5                                        Infeasible
    1      6     1.85127     -0.09197            1        0.109        0.778  start point
    2      9    0.300167         9.33            1       -0.117        0.313 Hessian modified
    3     12    0.529835       0.9209            1         0.12        0.232  twice
    4     16    0.186965       -1.517          0.5       -0.224         0.13
    5     19   0.0729085       0.3313            1       -0.121        0.054
    6     22   0.0353323     -0.03303            1      -0.0542       0.0271
    7     25   0.0235566     0.003184            1      -0.0271      0.00587
    8     28   0.0235504    9.032e-008           1      -0.0146     8.51e-007
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
                                     2
```

The output `history` is a structure that contains two fields:

```
history =

       x: [9x2 double]
    fval: [9x1 double]
```

The `fval` field contains the objective function values corresponding to the sequence of points computed by `fmincon`:

```
history.fval

ans =

    1.8394
    1.8513
    0.3002
    0.5298
```

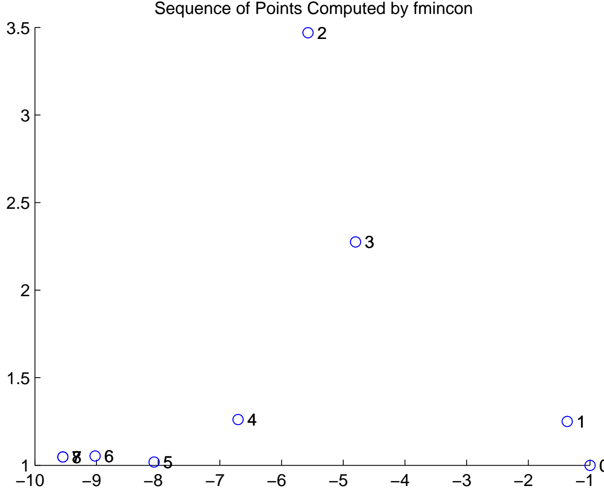
```
    0.1870
    0.0729
    0.0353
    0.0236
    0.0236
```

These are the same values displayed in the iterative output in the column with header `f(x)`.

The `x` field of `history` contains the sequence of points computed by the algorithm:

```
history.x

ans =

   -1.0000    1.0000
   -1.3679    1.2500
   -5.5708    3.4699
   -4.8000    2.2752
   -6.7054    1.2618
   -8.0679    1.0186
   -9.0230    1.0532
   -9.5471    1.0471
   -9.5474    1.0474
```

This example displays a plot of this sequence of points, in which each point is labeled by its iteration number.



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

The second output argument, `searchdir`, contains the search directions for `fmincon` at each iteration. The search direction is a vector pointing from the point computed at the current iteration to the point computed at the next iteration:

```
searchdir =

   -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
```

Default Options Settings

In this section...

"Introduction" on page 2-41

"Changing the Default Settings" on page 2-41

"Large-Scale vs. Medium-Scale Algorithms" on page 2-45

Introduction

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 "Main Algorithm" on page 4-74).

"Optimization Options" on page 7-7 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 toolbox functions. 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

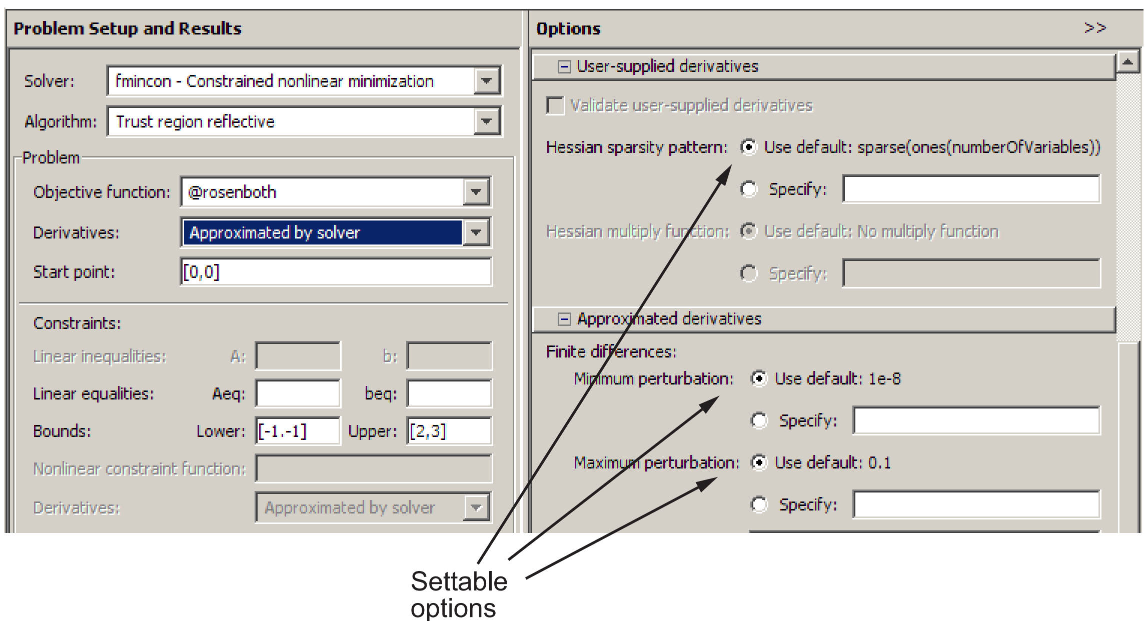
```
optimset fmincon
```

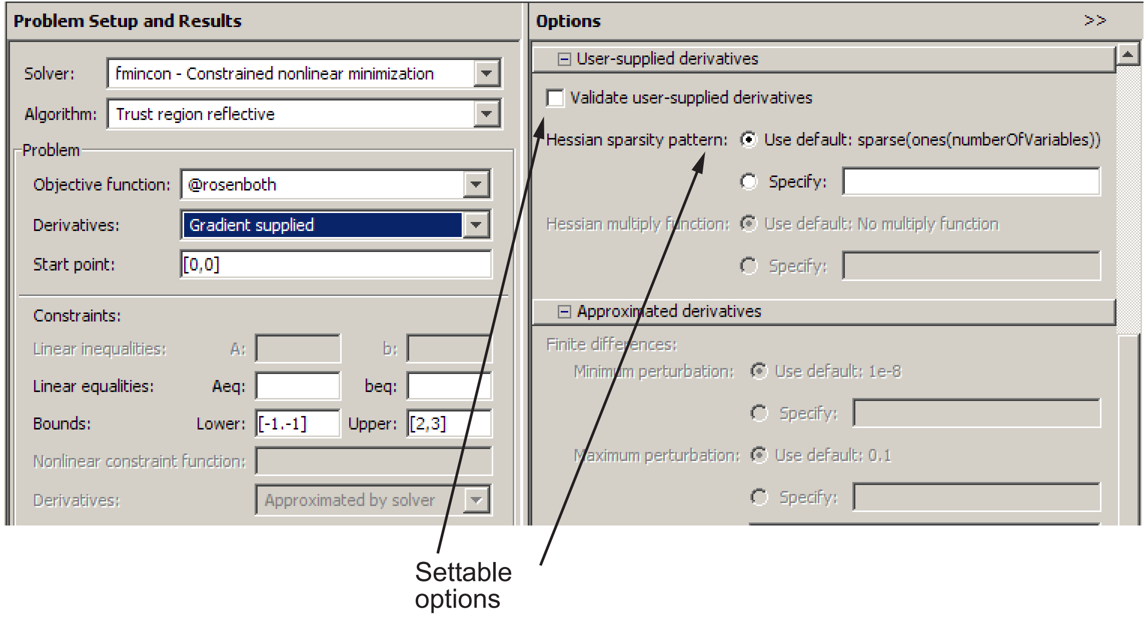
or

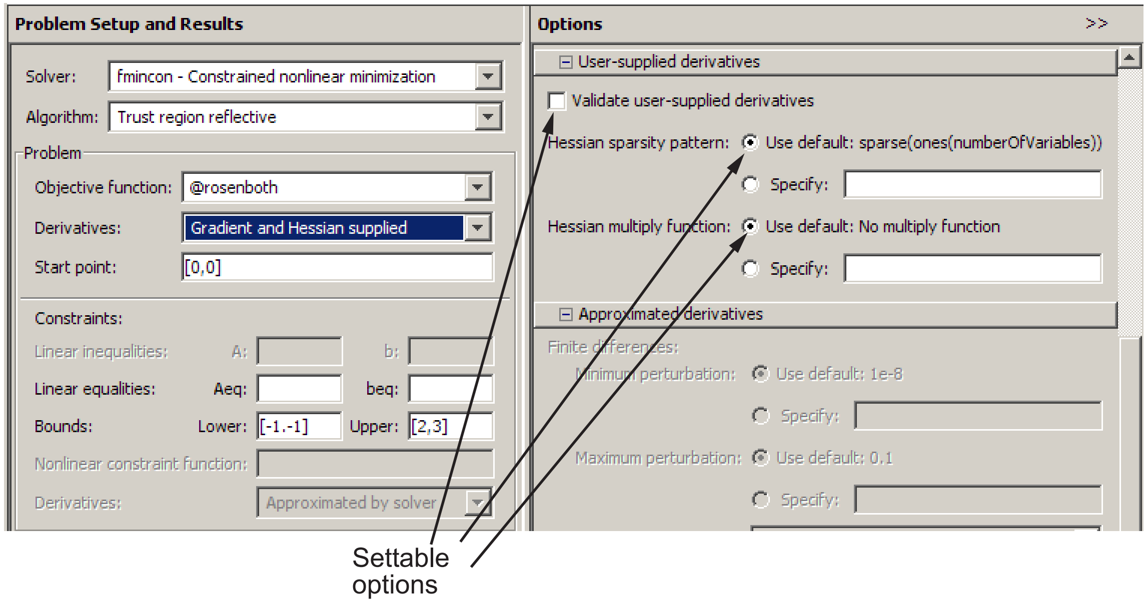
```
optimset(@fmincon)
```

This statement returns a structure. Generally, fields not used by the function have empty values (`[]`); fields used by the function are set to their default values for the given function. However, some solvers have different default values depending on the algorithm used. For example, `fmincon` has a default `MaxIter` value of 400 for the `trust-region-reflective` and `active-set` algorithms, but a default value of 1000 for the `interior-point` algorithm. `optimset fmincon` returns `[]` for the `MaxIter` field.

You can also check the available options and their defaults in the Optimization Tool. These can change depending on problem and algorithm settings. These three pictures show how the available options for derivatives change as the type of supplied derivatives change:


Problem Setup and Results
Solver: fmincon - Constrained nonlinear minimization
Algorithm: Trust region reflective
Problem
Objective function: @rosenboth
Derivatives: Approximated by solver
Start point: [0,0]
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: [-1,-1] Upper: [2,3]
Nonlinear constraint function:
Derivatives: Approximated by solver
Options
User-supplied derivatives
Validate user-supplied derivatives
Hessian sparsity pattern: Use default: sparse(ones(numberOfVariables))
Specify:
Hessian multiply function: Use default: No multiply function
Specify:
Approximated derivatives
Finite differences:
Minimum perturbation: Use default: 1e-8
Specify:
Maximum perturbation: Use default: 0.1
Specify:
Settable options


Problem Setup and Results
Solver: fmincon - Constrained nonlinear minimization
Algorithm: Trust region reflective
Problem
Objective function: @rosenboth
Derivatives: Gradient supplied
Start point: [0,0]
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: [-1,-1] Upper: [2,3]
Nonlinear constraint function:
Derivatives: Approximated by solver
Options
User-supplied derivatives
Validate user-supplied derivatives
Hessian sparsity pattern: Use default: sparse(ones(numberOfVariables))
Specify:
Hessian multiply function: Use default: No multiply function
Specify:
Approximated derivatives
Finite differences:
Minimum perturbation: Use default: 1e-8
Specify:
Maximum perturbation: Use default: 0.1
Specify:
Settable options



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 solver 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'`).

Choosing an Algorithm

Some solvers explicitly use an option called `LargeScale` for choosing which algorithm to use: `fminunc`, `linprog`, and others. Other solvers do not use the `LargeScale` attribute explicitly, but have an option called `Algorithm` instead: `fmincon`, `fsolve`, and others. All algorithms in `Algorithm` solvers are large scale, unless otherwise noted in their function reference pages. The term large-scale is explained in "Large-Scale vs. Medium-Scale Algorithms" on page 2-45. For all solvers that have a large-scale algorithm, the default

is for the function to use a large-scale algorithm (e.g., `LargeScale` is set to `'on'` by default).

To use a medium-scale algorithm in a solver that takes the `LargeScale` option, enter

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

For solvers that use the `Algorithm` option, choose the algorithm by entering

```
options = optimset('Algorithm','algorithm-name');
```

`algorithm-name` is the name of the chosen algorithm. You can find the choices in the function reference pages for each solver.

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 the following:

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

Large-Scale vs. Medium-Scale Algorithms

An optimization algorithm is *large scale* when it uses linear algebra that does not need to store, nor operate on, full matrices. This may be done internally by storing sparse matrices, and by using sparse linear algebra for computations whenever possible. Furthermore, the internal algorithms either preserve sparsity, such as a sparse Cholesky decomposition, or do

not generate matrices, such as a conjugate gradient method. Large-scale algorithms are accessed by setting the `LargeScale` option to `on`, or setting the `Algorithm` option appropriately (this is solver-dependent).

In contrast, *medium-scale* methods internally create full matrices and use dense linear algebra. If a problem is sufficiently large, full matrices take up a significant amount of memory, and the dense linear algebra may require a long time to execute. Medium-scale algorithms are accessed by setting the `LargeScale` option to `off`, or setting the `Algorithm` option appropriately (this is solver-dependent).

Don't let the name "large-scale" mislead you; you can use a large-scale algorithm on a small problem. Furthermore, you do not need to specify any sparse matrices to use a large-scale algorithm. Choose a medium-scale algorithm to access extra functionality, such as additional constraint types, or possibly for better performance.

Displaying Iterative Output

In this section...
"Introduction" on page 2-47
"Most Common Output Headings" on page 2-47
"Function-Specific Output Headings" on page 2-48

Note An optimization function does not return all of the output headings, described in the following tables, each time you call it. Which output headings are returned depends on the algorithm the optimization function uses for a particular problem.

Introduction

When you set `'Display'` to `'iter'` in `options`, the optimization functions display iterative output in the Command Window. This output, which provides information about the progress of the algorithm, is displayed in columns with descriptive headings. For example, if you run medium-scale `fminunc` with `'Display'` set to `'iter'`, the output headings are

```
                                                   First-order
 Iteration    Func-count  f(x)       Step-size     optimality
```

Most Common Output Headings

The following table lists some common output headings of iterative output.

Output Heading	Information Displayed
`Iteration` or `Iter`	Iteration number; see "Iterations and Function Counts" on page 2-27
`Func-count` or `F-count`	Number of function evaluations; see "Iterations and Function Counts" on page 2-27
`x`	Current point for the algorithm
`f(x)`	Current function value

Output Heading	Information Displayed
`Step-size`	Step size in the current search direction
`Norm of step`	Norm of the current step

Function-Specific Output Headings

The following sections describe output headings of iterative output whose meaning is specific to the optimization function you are using.

- "bintprog" on page 2-48
- "fminsearch" on page 2-49
- "fzero and fminbnd" on page 2-50
- "fminunc" on page 2-50
- "fsolve" on page 2-51
- "fgoalattain, fmincon, fminimax, and fseminf" on page 2-51
- "linprog" on page 2-52
- "lsqnonlin and lsqcurvefit" on page 2-53

bintprog

The following table describes the output headings specific to `bintprog`.

bintprog Output Heading	Information Displayed
`Explored nodes`	Cumulative number of explored nodes
`Obj of LP relaxation`	Objective function value of the linear programming (LP) relaxation problem
`Obj of best integer point`	Objective function value of the best integer point found so far. This is an upper bound for the final objective function value.

bintprog Output Heading	Information Displayed
`Unexplored nodes`	Number of nodes that have been set up but not yet explored
`Best lower bound on obj`	Objective function value of LP relaxation problem that gives the best current lower bound on the final objective function value
`Relative gap between bounds`	$\frac{100(b-a)}{\lvert b \rvert + 1}$, where • b is the objective function value of the best integer point. • a is the best lower bound on the objective function value.

fminsearch

The following table describes the output headings specific to `fminsearch`.

fminsearch Output Heading	Information Displayed
`min f(x)`	Minimum function value in the current simplex
`Procedure`	Simplex procedure at the current iteration. Procedures include `initial`, `expand`, `reflect`, `shrink`, `contract inside`, and `contract outside`. See "fminsearch Algorithm" on page 4-11 for explanations of these procedures.

fzero and fminbnd

The following table describes the output headings specific to `fzero` and `fminbnd`.

fzero and fminbnd Output Heading	Information Displayed
`Procedure`	Procedure at the current operation. Procedures for `fzero`: • `initial` (initial point) • `search` (search for an interval containing a zero) • `bisection` (bisection search) • `interpolation` Operations for `fminbnd`: • `initial` • `golden` (golden section search) • `parabolic` (parabolic interpolation)

fminunc

The following table describes the output headings specific to `fminunc`.

fminunc Output Heading	Information Displayed
`First-order optimality`	First-order optimality measure (see "First-Order Optimality Measure" on page 2-28)
`CG-iterations`	Number of conjugate gradient iterations taken by the current (optimization) iteration (see "Preconditioned Conjugate Gradient Method" on page 4-23)

fsolve

The following table describes the output headings specific to `fsolve`.

fsolve Output Heading	Information Displayed
`First-order optimality`	First-order optimality measure (see "First-Order Optimality Measure" on page 2-28)
`Trust-region radius`	Current trust-region radius (change in the norm of the trust-region radius)
`Residual`	Residual (sum of squares) of the function
`Directional derivative`	Gradient of the function along the search direction

fgoalattain, fmincon, fminimax, and fseminf

The following table describes the output headings specific to `fgoalattain`, `fmincon`, `fminimax`, and `fseminf`.

fgoalattain, fmincon, fminimax, fseminf Output Heading	Information Displayed
`Max constraint`	Maximum violation among all constraints, both internally constructed and user-provided
`First-order optimality`	First-order optimality measure (see "First-Order Optimality Measure" on page 2-28)
`CG-iterations`	Number of conjugate gradient iterations taken by the current (optimization) iteration (see "Preconditioned Conjugate Gradient Method" on page 4-23)
`Trust-region radius`	Current trust-region radius
`Residual`	Residual (sum of squares) of the function
`Attainment factor`	Value of the attainment factor for `fgoalattain`

fgoalattain, fmincon, fminimax, fseminf Output Heading	Information Displayed
`Objective value`	Objective function value of the nonlinear programming reformulation of the minimax problem for `fminimax`
`Directional derivative`	Current gradient of the function along the search direction
`Procedure`	Hessian update and QP subproblem. The `Procedure` messages are discussed in "Updating the Hessian Matrix" on page 4-29.

linprog

The following table describes the output headings specific to `linprog`.

linprog Output Heading	Information Displayed
`Primal Infeas A*x-b`	Primal infeasibility
`Dual Infeas A'*y+z-w-f`	Dual infeasibility
`Duality Gap x'*z+s'*w`	Duality gap (see "Large Scale Linear Programming" on page 4-74) between the primal objective and the dual objective. `s` and `w` appear only in this equation if there are finite upper bounds.
`Total Rel Error`	Total relative error, described at the end of "Main Algorithm" on page 4-74.
`Objective f'*x`	Current objective value

lsqnonlin and lsqcurvefit

The following table describes the output headings specific to `lsqnonlin` and `lsqcurvefit`.

lsqnonlin and lsqcurvefit Output Heading	Information Displayed
`Resnorm`	Value of the squared 2-norm of the residual at `x`
`Residual`	Residual vector of the function
`First-order optimality`	First-order optimality measure (see "First-Order Optimality Measure" on page 2-28)
`CG-iterations`	Number of conjugate gradient iterations taken by the current (optimization) iteration (see "Preconditioned Conjugate Gradient Method" on page 4-23)
`Directional derivative`	Gradient of the function along the search direction
`Lambda`	λ_k value defined in "Levenberg-Marquardt Method" on page 4-121. (This value is displayed when you use the Levenberg-Marquardt method and omitted when you use the Gauss-Newton method.)

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.

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.

Troubleshooting

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.

Troubleshooting (Continued)

Problem	Recommendation
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.
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.

Troubleshooting (Continued)

Problem	Recommendation
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 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. (From the MATLAB Help browser or the MathWorks™ Web site documentation, you can click the demo name to display the demo.)
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.

Local vs. Global Optima

What Are Local and Global Optima?

Usually, the goal of an optimization is to find a *local* minimum of a function—a point where the function value is smaller than at nearby points, but possibly greater than at a distant point in the search space. Sometimes the goal of an optimization is to find the *global* minimum—a point where the function value is smaller than all others in the search space. In general, optimization algorithms return a local minimum. This section describes why solvers behave this way, and gives suggestions for ways to search for a global minimum, if needed.

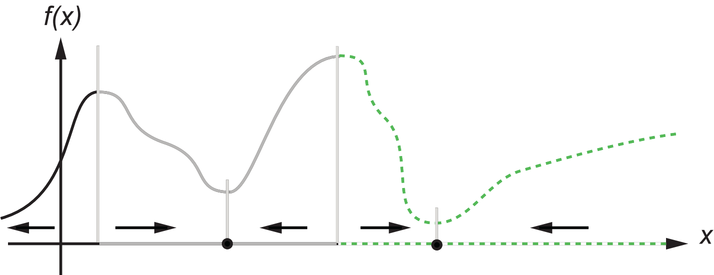
Basins of Attraction

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

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

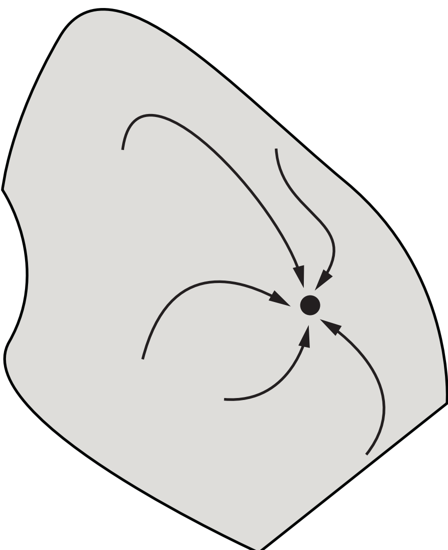
yields a path $x(t)$ that goes to a local minimum as t gets large. Generally, initial values $x(0)$ that are near to each other give steepest descent paths that tend to the same minimum point along their steepest descent paths. The set of initial values that lead to the same local minimum is called a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

The following figure shows two one-dimensional minima. Different basins of attraction are plotted with different line styles, and directions of steepest descent are indicated by arrows. For this and subsequent figures, black dots represent local minima. Every steepest descent path, starting at a point $x(0)$, goes to the black dot in the basin containing $x(0)$.



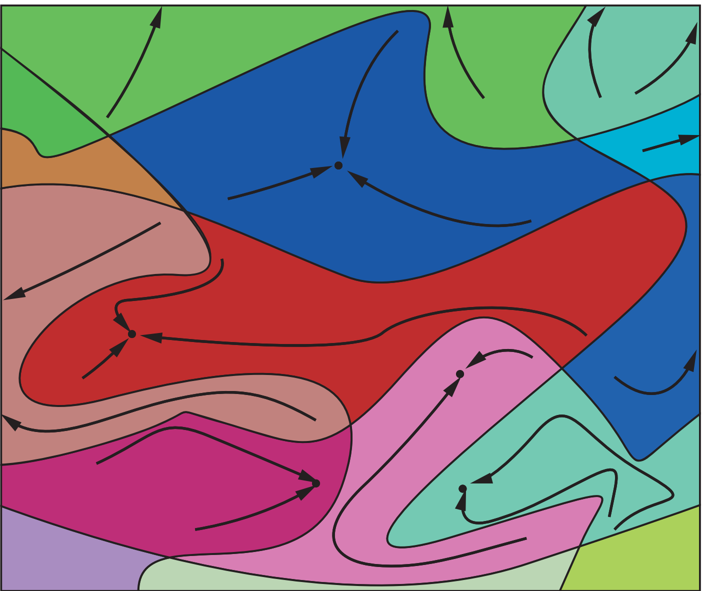
One-dimensional basins

The following figure shows how steepest descent paths can be more complicated in more dimensions.



One basin of attraction, showing steepest descent paths from various starting points

The following figure shows even more complicated paths and basins.



Several basins of attraction

Constraints can break up one basin of attraction into several pieces, where a steepest descent path may be restricted from proceeding.

Searching For Global Optima

Many numerical methods for optimization are based, in part, on the method of steepest descent.

Note Solvers do not precisely follow steepest descent paths. They attempt to take large steps, in the interest of efficiency. Basins of attraction associated with solver algorithms can be more complex than those of steepest descent.

The problem of global optimization turns into two parts:

- Finding a good initial value for optimizers
- Finding the minimum point in the basin associated with the initial value

---

Note Optimization Toolbox solvers generally find the minimum point in the basin, but leave the choice of starting point to you.

Generally, Optimization Toolbox solvers are not designed to find global optima. They find the optimum in the basin of attraction of the starting point. If you need a global optimum, you must find an initial value contained in the basin of attraction of a global optimum.

There are some exceptions to this general rule.

- Linear programming and positive definite quadratic programming problems are convex, with convex feasible regions, so there is only one basin of attraction. Indeed, under certain choices of options, `linprog` ignores any user-supplied starting point, and `quadprog` does not require one, though supplying one can sometimes speed a minimization.
- Multiobjective optimization does not have basins of attraction, but still depends on initial values.
- Some Genetic Algorithm and Direct Search Toolbox functions, such as `simulannealbnd`, are designed to search through more than one basin of attraction.

Suggestions for ways to set initial values to search for a global optimum:

- Use a regular grid of initial points.
- Use random points drawn from a uniform distribution if your problem has all its coordinates bounded, or from normal, exponential, or other random distributions if some components are unbounded. The less you know about the location of the global optimum, the more spread-out your random distribution should be. For example, normal distributions rarely sample more than three standard deviations away from their means, but a Cauchy distribution (density $1/(\pi(1 + x^2))$) makes hugely disparate samples.
- Use identical initial points with added random perturbations on each coordinate, bounded, normal, exponential, or other.

- Use the Genetic Algorithm and Direct Search Toolbox function `gacreationlinearfeasible` to obtain a set of random initial points in a region with linear constraints.

The more you know about possible initial points, the more focused and successful your search will be.

Reference

[1] Nocedal, Jorge and Wright, Stephen J. *Numerical Optimization*, Second Edition. New York: Springer, 2006.

Optimization Tool

- “Getting Started with the Optimization Tool” on page 3-2
- “Running a Problem in the Optimization Tool” on page 3-6
- “Specifying Certain Options” on page 3-10
- “Getting Help in the Optimization Tool” on page 3-13
- “Importing and Exporting Your Work” on page 3-14
- “Optimization Tool Examples” on page 3-18

Getting Started with the Optimization Tool

In this section...

"Introduction" on page 3-2
"Opening the Optimization Tool" on page 3-2
"Steps for Using the Optimization Tool" on page 3-5

Introduction

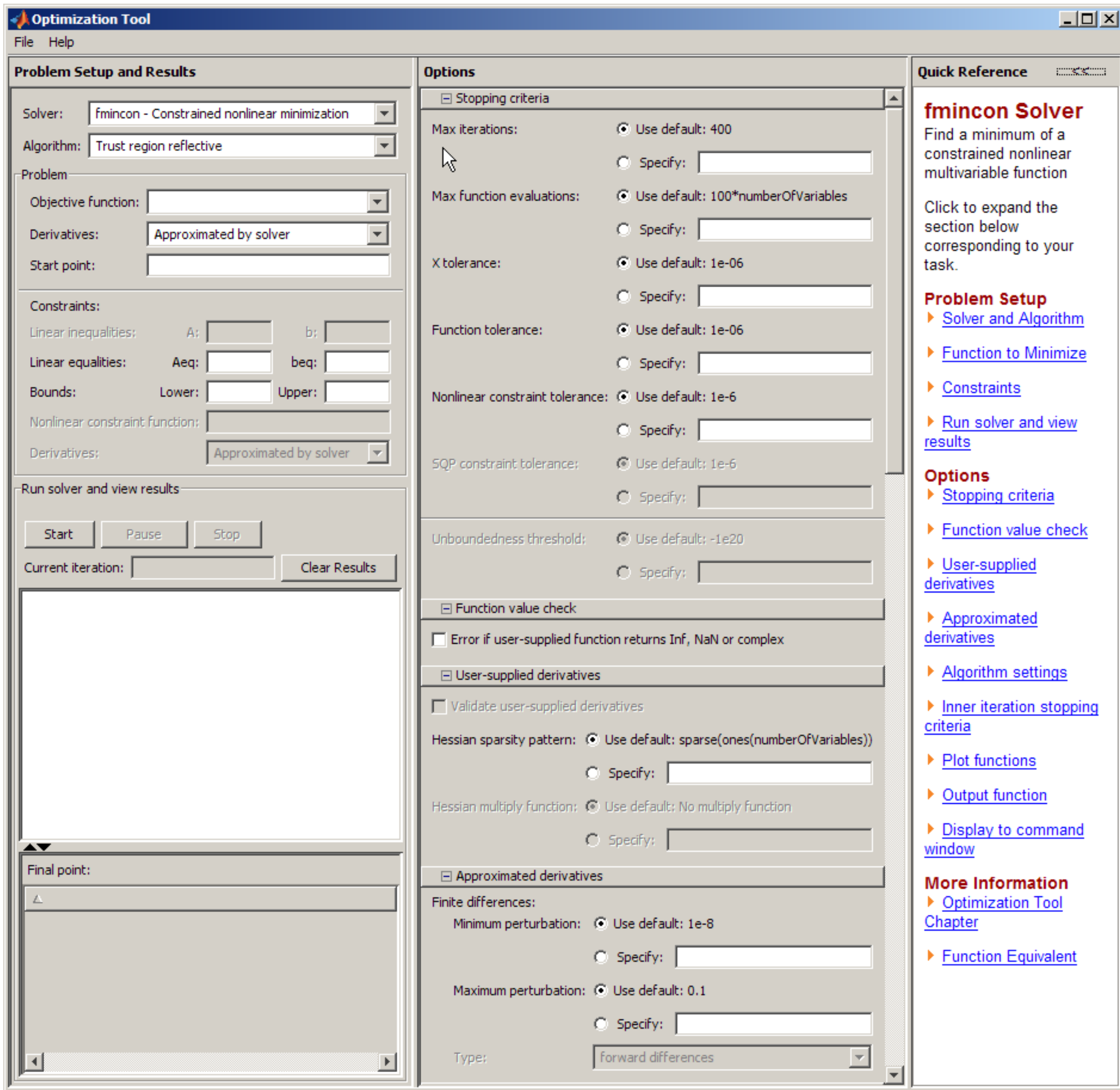
The Optimization Tool is a GUI for solving optimization problems. With the Optimization Tool, you select a solver from a list and set up your problem visually. If you are familiar with the optimization problem you want to solve, the Optimization Tool lets you select a solver, specify the optimization options, and run your problem. You can also import and export data from the MATLAB workspace, and generate M-files containing your configuration for the solver and options.

Opening the Optimization Tool

To open the tool, type

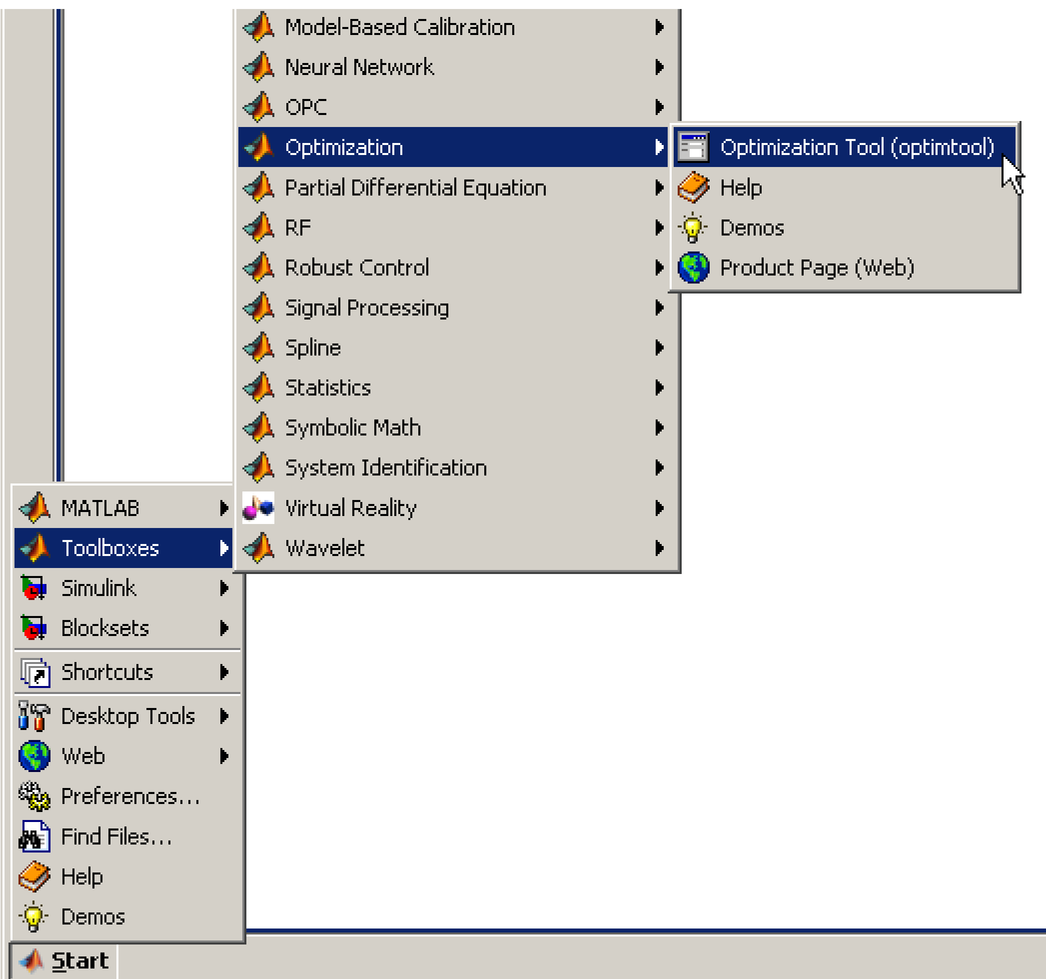
```
optimtool
```

in the Command Window. This opens the Optimization Tool, as shown in the following figure.



Optimization Tool
File Help
Problem Setup and Results
Solver: fmincon - Constrained nonlinear minimization
Algorithm: Trust region reflective
Problem
Objective function:
Derivatives: Approximated by solver
Start point:
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function:
Derivatives: Approximated by solver
Run solver and view results
Start Pause Stop
Current iteration: Clear Results
Final point:
Options
Stopping criteria
Max iterations: Use default: 400
Specify:
Max function evaluations: Use default: 100*numberOfVariables
Specify:
X tolerance: Use default: 1e-06
Specify:
Function tolerance: Use default: 1e-06
Specify:
Nonlinear constraint tolerance: Use default: 1e-6
Specify:
SQP constraint tolerance: Use default: 1e-6
Specify:
Unboundedness threshold: Use default: -1e20
Specify:
Function value check
Error if user-supplied function returns Inf, NaN or complex
User-supplied derivatives
Validate user-supplied derivatives
Hessian sparsity pattern: Use default: sparse(ones(numberOfVariables))
Specify:
Hessian multiply function: Use default: No multiply function
Specify:
Approximated derivatives
Finite differences:
Minimum perturbation: Use default: 1e-8
Specify:
Maximum perturbation: Use default: 0.1
Specify:
Type: forward differences
Quick Reference
fmincon Solver
Find a minimum of a constrained nonlinear multivariable function
Click to expand the section below corresponding to your task.
Problem Setup
Solver and Algorithm
Function to Minimize
Constraints
Run solver and view results
Options
Stopping criteria
Function value check
User-supplied derivatives
Approximated derivatives
Algorithm settings
Inner iteration stopping criteria
Plot functions
Output function
Display to command window
More Information
Optimization Tool Chapter
Function Equivalent

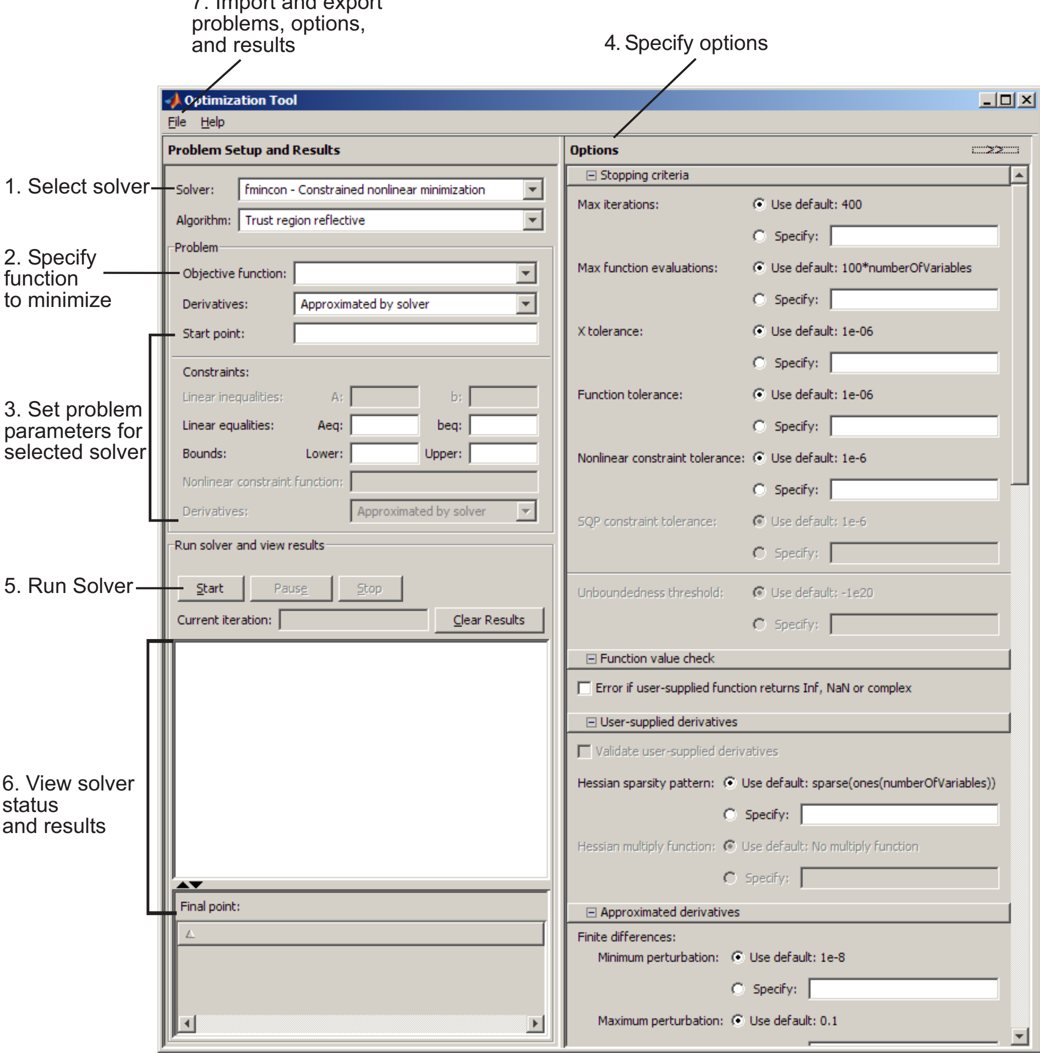
You can also open the Optimization Tool from the main MATLAB window as pictured:



The reference page for the Optimization Tool provides variations for starting the `optimtool` function.

Steps for Using the Optimization Tool

This is a summary of the steps to set up your optimization problem and view results with the Optimization Tool.



Running a Problem in the Optimization Tool

In this section...

"Introduction" on page 3-6

"Pausing and Stopping the Algorithm" on page 3-7

"Viewing Results" on page 3-7

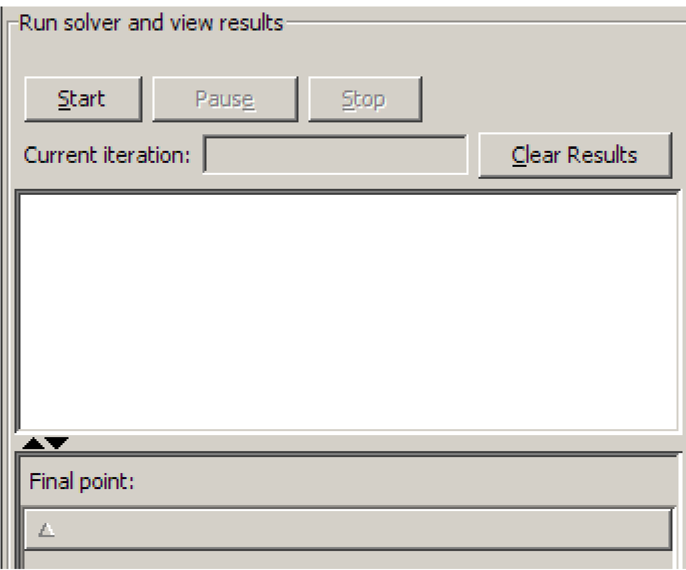
"Final Point" on page 3-7

"Starting a New Problem" on page 3-8

"Closing the Optimization Tool" on page 3-9

Introduction

After defining your problem and specifying the options, you are ready to run the solver.



To run the selected solver, click the **Start** button. For most solvers, as the algorithm runs, the **Current iteration** field updates. This field does not update for solvers for which the current iteration does not apply.

Pausing and Stopping the Algorithm

While the algorithm is running, you can

- Click **Pause** to temporarily suspend the algorithm. To resume the algorithm using the current iteration at the time you paused, click **Resume**.
- Click **Stop** to stop the algorithm. The **Run solver and view results** window displays information for the current iteration at the moment you clicked **Stop**.

You can export your results after stopping the algorithm. For details, see "Exporting to the MATLAB Workspace" on page 3-14.

Viewing Results

When the algorithm terminates, the **Run solver and view results** window displays the reason the algorithm terminated. To clear the **Run solver and view results** window between runs, click **Clear Results**.

Displaying Plots

In addition to the **Run solver and view results** window, you can also display measures of progress while the algorithm executes by generating plots. Each plot selected draws a separate axis in the figure window. You can select a predefined plot function from the Optimization Tool, or you can write your own. For more information on what plot functions are available, see "Plot Functions" on page 3-10.

Final Point

The **Final point** updates to show the coordinates of the final point when the algorithm terminated. If you don't see the final point, click the upward-pointing triangle on the icon on the lower-left.

Starting a New Problem

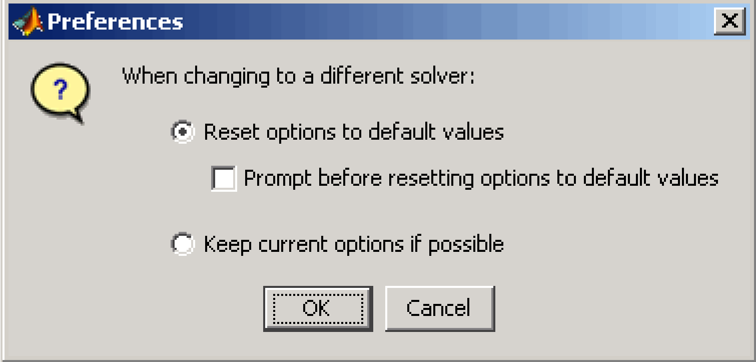
Resetting Options and Clearing the Problem

Selecting **File > Reset Optimization Tool** resets the problem definition and options to the original default values. This action is equivalent to closing and restarting the optimtool.

To clear only the problem definition, select **File > Clear Problem Fields**. With this action, fields in the **Problem Setup and Results** pane are reset to the defaults, with the exception of the selected solver and algorithm choice. Any options that you have modified from the default values in the **Options** pane are not reset with this action.

Setting Preferences for Changing Solvers

To modify how your options are handled in the Optimization Tool when you change solvers, select **File > Preferences**, which opens the Preferences dialog box shown below.



The default value, **Reset options to defaults**, discards any options you specified previously in the `optimtool`. Under this choice, you can select the option **Prompt before resetting options to defaults**.

Alternatively, you can select **Keep current options if possible** to preserve the values you have modified. Changed options that are not valid with the newly selected solver are kept but not used, while active options relevant

to the new solver selected are used. This choice allows you to try different solvers with your problem without losing your options.

Closing the Optimization Tool

To close the `optimtool` window, select **File > Close**.

Specifying Certain Options

In this section...
“Plot Functions” on page 3-10
“Output function” on page 3-11
“Display to Command Window” on page 3-11

Plot Functions

You can select a plot function to easily plot various measures of progress while the algorithm executes. Each plot selected draws a separate axis in the figure window. If available for the solver selected, the **Stop** button in the **Run solver and view results** window to interrupt a running solver. You can select a predefined plot function from the Optimization Tool, or you can select **Custom function** to write your own. Plot functions not relevant to the solver selected are grayed out. The following lists the available plot functions:

- **Current point** — Select to show a bar plot of the point at the current iteration.
- **Function count** — Select to plot the number of function evaluations at each iteration.
- **Function value** — Select to plot the function value at each iteration.
- **Norm of residuals** — Select to show a bar plot of the current norm of residuals at the current iteration.
- **Max constraint** — Select to plot the maximum constraint violation value at each iteration.
- **Current step** — Select to plot the algorithm step size at each iteration.
- **First order optimality** — Select to plot the violation of the optimality conditions for the solver at each iteration.
- **Custom function** — Enter your own plot function as a function handle. To provide more than one plot function use a cell array, for example, by typing:

  ```
  {@plotfcn,@plotfcn2}
  ```

  See “Plot Functions” on page 7-26.

Plot functions

Current point | Function count | Function value

Max constraint | Current step | First order optimality

Custom function:

The graphic above shows the plot functions available for the default `fmincon` solver.

Output function

Output function is a function or collection of functions the algorithm calls at each iteration. Through an output function you can observe optimization quantities such as function values, gradient values, and current iteration. Specify no output function, a single output function using a function handle, or multiple output functions. To provide more than one output function use a cell array of function handles in the **Custom function** field, for example by typing:

```
{@outputfcn,@outputfcn2}
```

For more information on writing an output function, see "Output Function" on page 7-17.

Output function

Custom function:

Display to Command Window

Select **Level of display** to specify the amount of information displayed when you run the algorithm. Choose from the following:

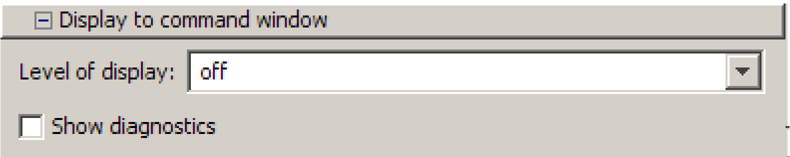
- `off` (default) — Display no output.
- `final` — Display only the reason for stopping at the end of the run.
- `notify` — Display output only if the function does not converge.

- `iterative` — Display information at each iteration of the algorithm.

Set **Node interval**, with the `bintprog` solver selected, to specify the interval of explored nodes you want to display output for. Note that integer feasible solution nodes are always shown.

Selecting **Show diagnostics** lists problem information and options that have changed from the defaults.

The graphic below shows the display options.



Getting Help in the Optimization Tool

In this section...

"Quick Reference" on page 3-13
"Additional Help" on page 3-13

Quick Reference

The Optimization Tool provides extensive context-sensitive help directly in the GUI.

For assistance with the primary tasks in the Optimization Tool window, use the **Quick Reference** pane. To toggle between displaying or hiding the **Quick Reference** pane, do either of the following:

- Select **Help > Show Quick Reference**
- Click the >> or << buttons in the upper right of the GUI

To resize the **Quick Reference** pane, drag the vertical divider to the left or to the right.

Additional Help

In addition to the **Quick Reference** pane, you can access the documentation for the Optimization Tool by selecting **Help > Optimization Tool Help**.

Importing and Exporting Your Work

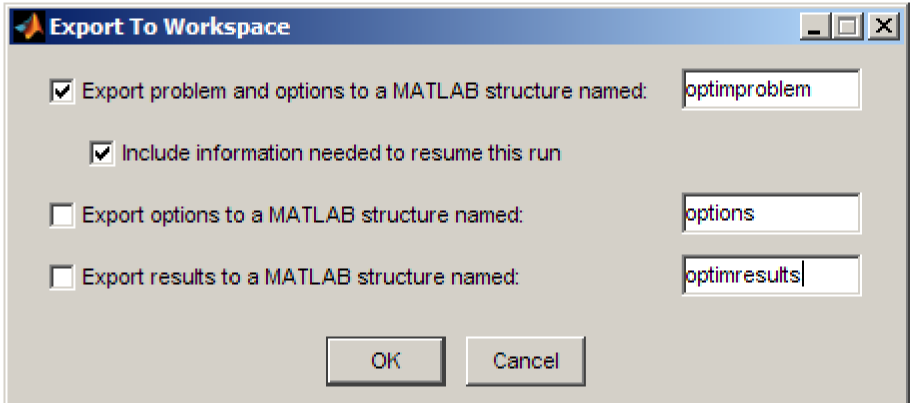
In this section...

"Exporting to the MATLAB Workspace" on page 3-14
"Importing Your Work" on page 3-16
"Generating an M-File" on page 3-16

Exporting to the MATLAB Workspace

The **Export to Workspace** dialog box enables you to send your problem information to the MATLAB workspace as a structure that you may then manipulate in the Command Window.

To access the **Export to Workspace** dialog box shown below, select **File > Export to Workspace**.



You can specify a structure that contains:

- The problem and options information
- The problem and options information, and the state of the solver when stopped (this means the latest point for most solvers, the current population for Genetic Algorithms solvers, and the best point found for Simulated Annealing and Threshold Acceptance solvers)

- The states of random number generators `rand` and `randn` at the start of the previous run, by checking the **Use random states from previous run** box for applicable solvers
- The options information only
- The results of running your problem in `optimtool`

Exported results structures contain all optional information. For example, an exported results structure for `lsqcurvefit` contains the data `x, resnorm, residual, exitflag, output, lambda,` and `jacobian`.

After you have exported information from the Optimization Tool to the MATLAB workspace, you can see your data in the MATLAB Workspace browser or by typing the name of the structure at the Command Window. To see the value of a field in a structure, double-click on the structure in the Workspace window. Alternatively, see the values by entering `structurename.fieldname` at the command line. For example, so see the message in an output structure, enter `output.message`. If a structure contains structures, you can double-click again in the workspace browser, or enter `structure1.structure2.fieldname` at the command line. For example, to see the level of iterative display contained in the options structure of an exported problem structure, enter `optimproblem.options.Display`.

You can run a solver on an exported problem at the command line by typing

```
solver(problem)
```

For example, if you have exported a `fmincon` problem named `optimproblem`, you can type

```
fmincon(optimproblem)
```

This runs `fmincon` on the problem with the saved options structure contained in `optimproblem`. You can exercise more control over outputs by typing, for example,

```
[x,fval,exitflag] = fmincon(optimproblem)
```

or use any other supported syntax.

Importing Your Work

Whether you saved options from Optimization Toolbox functions at the Command Window or if you exported options, or the problem and options, from the `optimtool`, you can resume work on your problem using the `optimtool`.

There are three ways to import your options, or problem and options, to `optimtool`.

- Call the `optimtool` function from the Command Window specifying your options, or problem and options, as the input, tor example,

```
optimtool(options)
```

- Select **File > Import Options** in the Optimization Tool.
- Select **File > Import Problem** in the Optimization Tool.

The methods described above require that the options, or problem and options, be present in the MATLAB workspace.

If you import a problem that was generated with the **Include information needed to resume this run** box checked, the initial point is the latest point generated in the previous run. (For Genetic Algorithm solvers, the initial population is the latest population generated in the previous run. For Simulated Annealing and Threshold Acceptance solvers, the initial point is the best point generated in the previous run.) If you import a problem that was generated with this box unchecked, the initial point (or population) is the initial point (or population) of the previous run.

Generating an M-File

You may want to generate an M-file to continue with your optimization problem in the Command Window at another time. You can run the M-file without modification to recreate the results that you created with the Optimization Tool. You can also edit and modify the M-file and run it from the Command Window.

To export data from the Optimization Tool to an M-file, select **File > Generate M-file**.

The M-file captures the following:

- The problem definition, including the solver, information on the function to be minimized, algorithm specification, constraints, and start point
- The options (using `optimset`) with the currently selected option value

Running the M-file at the Command Window reproduces your problem results.

Although you cannot export your problem results to a generated M-file, you can save them in a MAT-file that you can use with your generated M-file, by exporting the results using the Export to Workspace dialog box, then saving the data to a MAT-file from the Command Window.

Optimization Tool Examples

In this section...

"About Optimization Tool Examples" on page 3-18

"Optimization Tool with the fmincon Solver" on page 3-18

"Optimization Tool with the lsqlin Solver" on page 3-22

About Optimization Tool Examples

This section contains two examples showing how to use the Optimization Tool to solve representative problems. There are other examples available: "Problem Formulation: Rosenbrock's Function" on page 1-4 and "Example: Constrained Minimization Using fmincon's Interior-Point Algorithm With Analytic Hessian" on page 4-50 in this User's Guide, and several in the Genetic Algorithm and Direct Search Toolbox User's Guide.

Optimization Tool with the fmincon Solver

This example shows how to use the Optimization Tool with the `fmincon` solver to minimize a quadratic subject to linear and nonlinear constraints and bounds.

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

$$\min_x f(x) = {x_1}^2 + {x_2}^2$$

subject to the constraints

$$\begin{aligned} 0.5 &\leq x_1 \\ -x_1 - x_2 + 1 &\leq 0 \\ -{x_1}^2 - {x_2}^2 + 1 &\leq 0 \\ -9{x_1}^2 - {x_2}^2 + 9 &\leq 0 \\ -{x_1}^2 - x_2 &\leq 0 \\ -{x_2}^2 + x_1 &\leq 0 \end{aligned}$$

The starting guess for this problem is $x_1 = 3$ and $x_2 = 1$.

Step 1: Write an M-file objfun.m for the objective function.

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

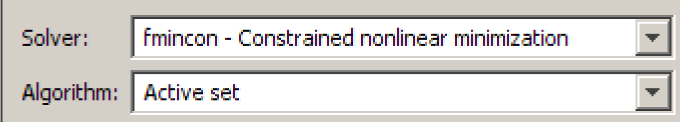
Step 2: Write an M-file nonlconstr.m for the constraints.

```
function [c,ceq] = nonlconstr(x)
c = [-x(1)^2 - x(2)^2 + 1;
     -9*x(1)^2 - x(2)^2 + 9;
     -x(1)^2 + x(2);
     -x(2)^2 + x(1)];
ceq = [];
```

Step 3: Set up and run the problem with the Optimization Tool.

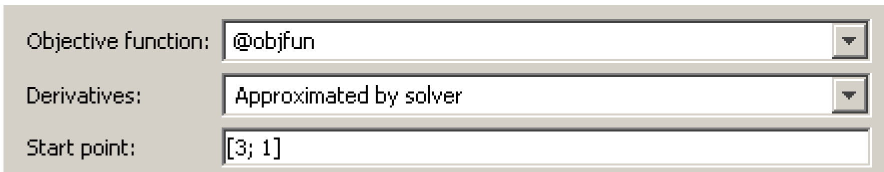
1. Enter `optimtool` in the Command Window to open the Optimization Tool.

2. Select `fmincon` from the selection of solvers and change the **Algorithm** field to `Active set`.



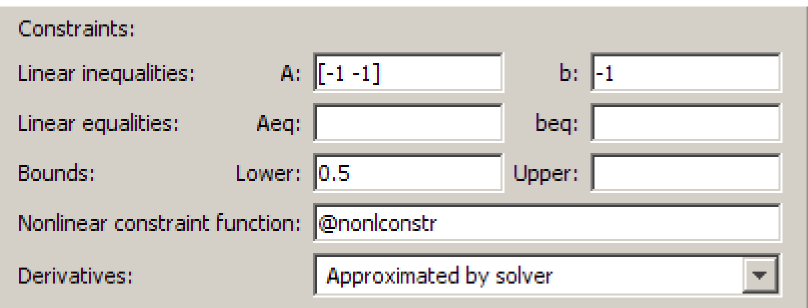
3. Enter `@objfun` in the **Objective function** field to call the M-file `objfun.m`.

4. Enter `[3; 1]` in the **Start point** field.

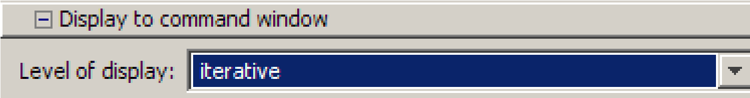


5. Define the constraints.

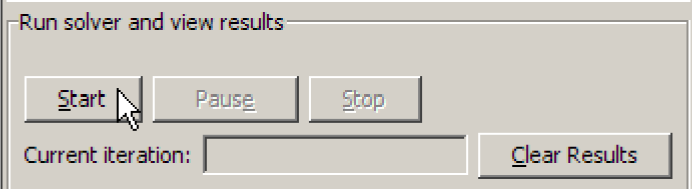
- To create variables for the equality constraints, enter `[-1 -1]` in the **A** field and enter `-1`in the **b** field.
- Set the bounds on the variables to be $0.5 \le x_1$ by entering `0.5` for **Lower**.
- Enter `@nonlconstr` in the **Nonlinear constraint function** field to call the M-file `nonlconstr.m`.



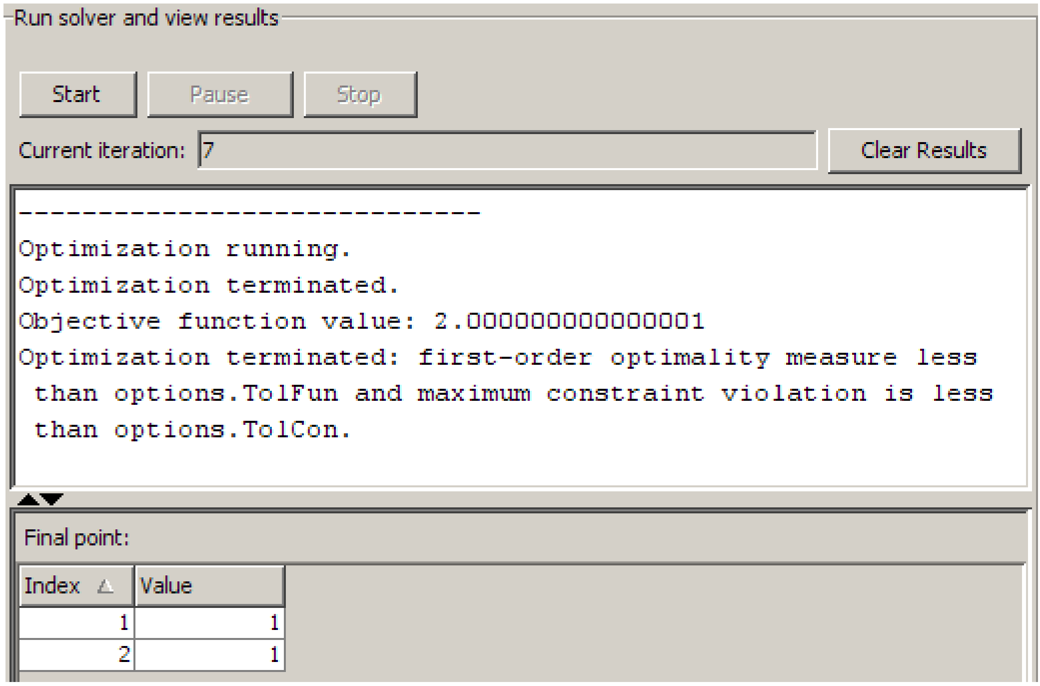
6 In the **Options** pane, expand the **Display to command window** option if necessary, and select `Iterative` to show algorithm information at the Command Window for each iteration.



7 Click the **Start** button as shown in the following figure.



8 When the algorithm terminates, under **Run solver and view results** the following information is displayed:



- The **Current iteration** value when the algorithm terminated, which for this example is `7`.
- The final value of the objective function when the algorithm terminated:

```
Objective function value: 2.000000000000001
```

- The algorithm termination message:

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
```

- The final point, which for this example is

```
1
1
```

9 In the Command Window, the algorithm information is displayed for each iteration:

```
                              max    Line search  Directional First-order
Iter F-count      f(x)  constraint  steplength   derivative  optimality
   0      3        10           2
   1      6   4.84298     -0.1322           1         -3.4        1.74
   2      9    4.0251    -0.01168           1        -0.78        4.08
   3     12   2.42704    -0.03214           1        -1.37        1.09
   4     15   2.03615   -0.004728           1       -0.373       0.995
   5     18   2.00033 -5.596e-005           1      -0.0357      0.0664
   6     21         2 -5.327e-009           1    -0.000326    0.000522
   7     24         2  -2.22e-016           1   -2.69e-008   1.21e-008
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
                                      3
                                      4
```

Reference

[1] Schittkowski, K., "More Test Examples for Nonlinear Programming Codes," *Lecture Notes in Economics and Mathematical Systems*, Number 282, Springer, p. 45, 1987.

Optimization Tool with the lsqlin Solver

This example shows how to use the Optimization Tool to solve a constrained least-squares problem.

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:

$$\min_x f(x) = x_1^2 + x_2^2 + x_3^2,$$

subject to the constraint

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

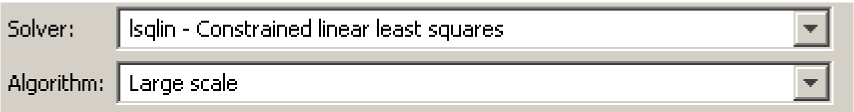
The function $f(x)$ is called the *objective function* and $x_1 + 2x_2 + 4x_3 = 7$ is an *equality constraint*. More complicated problems might contain other equality constraints, inequality constraints, and upper or lower bound constraints.

Setting Up the Problem

This section shows how to set up the problem with the `lsqlin` solver in the Optimization Tool.

1 Enter `optimtool` in the Command Window to open the Optimization Tool.

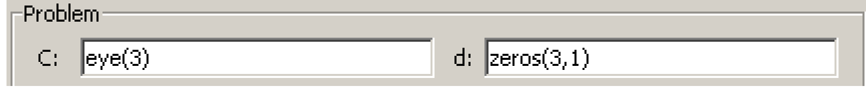
2 Select `lsqlin` from the selection of solvers. Use the default large-scale algorithm.



3 Enter the following to create variables for the objective function:

- In the **C** field, enter `eye(3)`.
- In the **d** field, enter `zeros(3,1)`.

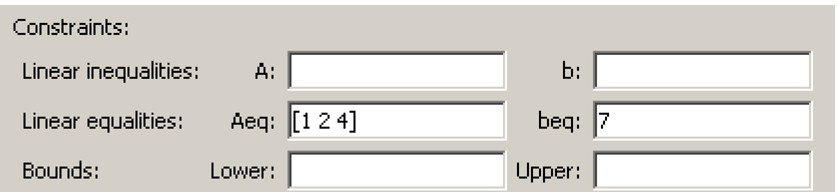
The **C** and **d** fields should appear as shown in the following figure.



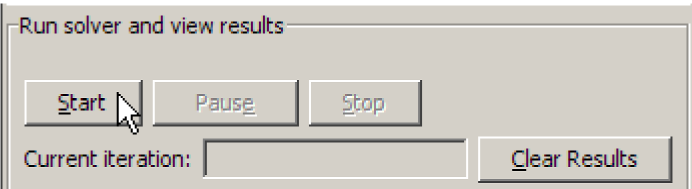
4 Enter the following to create variables for the equality constraints:

- In the **Aeq** field, enter `[1 2 4]`.
- In the **beq** field, enter `7`.

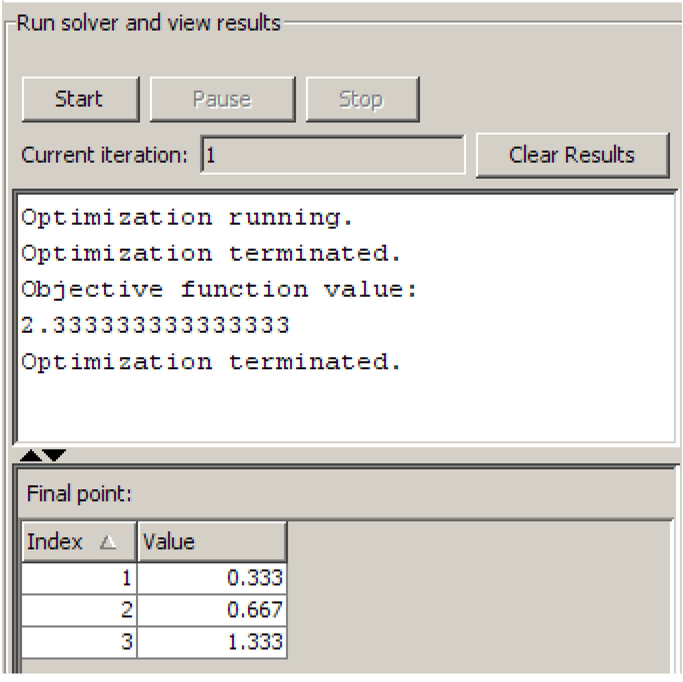
The **Aeq** and **beq** fields should appear as shown in the following figure.



5 Click the **Start** button as shown in the following figure.



6 When the algorithm terminates, under **Run solver and view results** the following information is displayed:



- The **Current iteration** value when the algorithm terminated, which for this example is `1`.
- The final value of the objective function when the algorithm terminated:

```
Objective function value: 2.333333333333333
```

- The algorithm termination message:

```
Optimization terminated.
```

- The final point, which for this example is

```
    0.3333
    0.6667
    1.3333
```

4

Using Optimization Toolbox Solvers

- "Optimization Theory Overview" on page 4-2
- "Unconstrained Nonlinear Optimization" on page 4-3
- "Unconstrained Nonlinear Optimization Examples" on page 4-14
- "Constrained Nonlinear Optimization" on page 4-20
- "Constrained Nonlinear Optimization Examples" on page 4-44
- "Linear Programming" on page 4-74
- "Linear Programming Examples" on page 4-86
- "Quadratic Programming" on page 4-90
- "Quadratic Programming Examples" on page 4-100
- "Binary Integer Programming" on page 4-108
- "Binary Integer Programming Example" on page 4-111
- "Least Squares (Model Fitting)" on page 4-116
- "Least Squares (Model Fitting) Examples" on page 4-126
- "Multiobjective Optimization" on page 4-141
- "Multiobjective Optimization Examples" on page 4-147
- "Equation Solving" on page 4-154
- "Equation Solving Examples" on page 4-162
- "Selected Bibliography" on page 4-172

Optimization Theory Overview

Optimization techniques are used to find a set of design parameters, $x = \{x_1,x_1,...,x_n\}$, 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, $G_i(x) = 0$ ($i = 1,...,m_e$); inequality constraints, $G_i(x) \leq 0$ ($i = m_e + 1,...,m$); and/or parameter bounds, x_l, x_u.

A General Problem (GP) description is stated as

$$\min_{x} f(x), \tag{4-1}$$

subject to

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

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 Nonlinear Optimization

In this section...

"Definition" on page 4-3

"Large Scale fminunc Algorithm" on page 4-3

"Medium Scale fminunc Algorithm" on page 4-6

"fminsearch Algorithm" on page 4-11

Definition

Unconstrained minimization is the problem of finding a vector x that is a local minimum to a scalar function $f(x)$:

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

The term *unconstrained* means that no restriction is placed on the range of x.

Large Scale fminunc Algorithm

Trust-Region Methods for Nonlinear Minimization

Many of the methods used in Optimization Toolbox solvers 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,

$$\min_s \{q(s),\ s \in N\}. \tag{4-2}$$

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

$$\min\left\{\frac{1}{2}s^T Hs + s^T g \ \text{ such that } \ \|Ds\| \le \Delta\right\}, \tag{4-3}$$

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 $\|\ .\ \|$ is the 2-norm. Good algorithms exist for solving Equation 4-3 (see [48]); 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 Equation 4-3. 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 Equation 4-3, have been proposed in the literature ([42] and [50]). The approximation approach followed in Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). Once the subspace S has been computed, the work to solve Equation 4-3 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 solver defines S as the linear space spanned by s_1 and s_2, 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

$$H \cdot s_2 = -g, \tag{4-4}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{4-5}$$

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 sketch of unconstrained minimization using trust-region ideas is now easy to give:

1. Formulate the two-dimensional trust-region subproblem.
2. Solve Equation 4-3 to determine the trial step s.
3. If $f(x + s) < f(x)$, then $x = x + s$.
4. 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 [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat 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 Gradient Method

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.

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., $d^T Hd \leq 0$. 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 4-3.

Medium Scale fminunc Algorithm

Basics of 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 [30]) are most suitable for problems that are not smooth 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

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2. \quad \textbf{(4-6)}$$

The minimum of this function is at $x = [1,1]$, where $f(x) = 0$. A contour map of this function is shown in the figure below, 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.

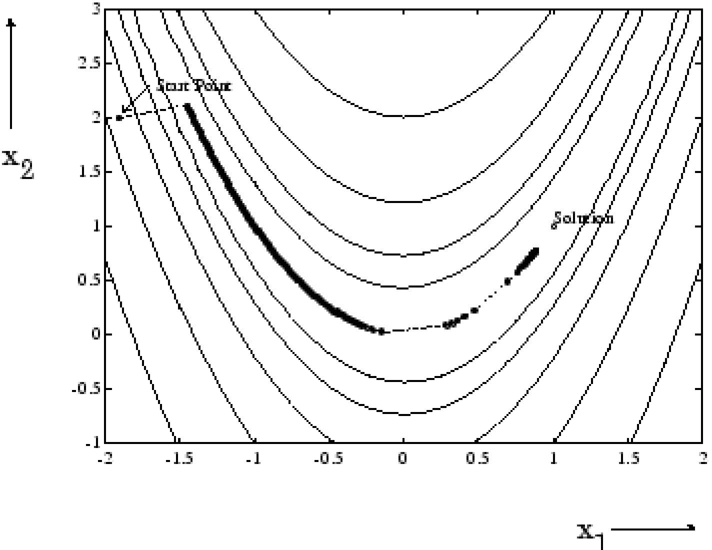


Figure 4-1: Steepest Descent Method on Rosenbrock's Function (Eq. 3-2)

This function, also known as the banana function, is notorious in unconstrained examples because of the way the curvature bends around the origin. Rosenbrock's function 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.

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

$$\min_{x} \frac{1}{2} x^T H x + c^T x + b, \tag{4-7}$$

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

$$\nabla f(x^*) = Hx^* + c = 0. \tag{4-8}$$

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

$$x^* = -H^{-1}c. \tag{4-9}$$

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 [12], Goldfarb [20], and Shanno [37] (BFGS) is thought to be the most effective for use in a general purpose method.

The formula given by BFGS is

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T s_k^T s_k H_k}{s_k^T H_k s_k}, \tag{4-10}$$

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 [7], Fletcher, and Powell [14]. This uses the same formula as the BFGS method (Equation 4-10) 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

$$d = -H_k^{-1} \cdot \nabla f(x_k). \tag{4-11}$$

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function in Figure 4-2, 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.

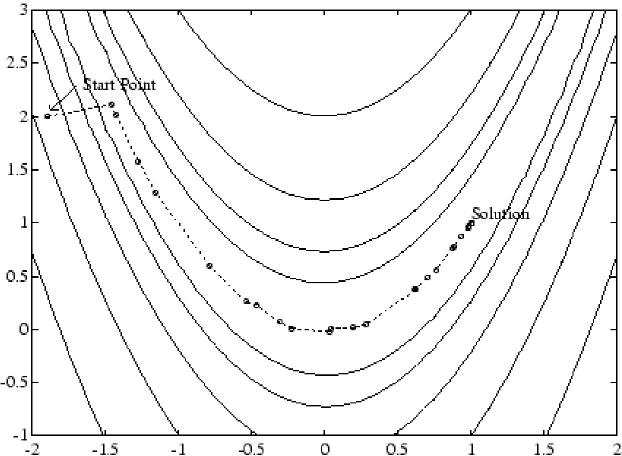


Figure 4-2: BFGS Method on Rosenbrock's Function

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

$$x_{k+1} = x_k + \alpha * d_k, \tag{4-12}$$

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 $x_k + \alpha^* d_k$ 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:

$$f(x_k + \alpha d_k) \le f(x_k) + c_1 \alpha \nabla f_k^T d_k, \tag{4-13}$$

$$\nabla f(x_k + \alpha d_k)^T d_k \ge c_2 \nabla f_k^T d_k, \tag{4-14}$$

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

The first condition (Equation 4-13) requires that α_k sufficiently decreases the objective function. The second condition (Equation 4-14) ensures that the step length is not too small. Points that satisfy both conditions (Equation 4-13 and Equation 4-14) are called *acceptable points*.

The line search method is an implementation of the algorithm described in Section 2-6 of [13]. See also [31] for more information about line search.

Hessian Update

Many of the optimization functions determine the direction of search by updating the Hessian matrix at each iteration, using the BFGS method (Equation 4-10). The function `fminunc` also provides an option to use the DFP method given in "Quasi-Newton Methods" on page 4-8 (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 Equation 4-15) 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,

$$q_k^T s_k = \alpha_k \left(\nabla f\left(x_{k+1}\right)^T d - \nabla f\left(x_k\right)^T d \right). \tag{4-15}$$

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 the negative gradient $-\nabla f(x_k)^T d$ 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.

fminsearch Algorithm

`fminsearch` uses the Nelder-Mead simplex algorithm as described in [57]. This algorithm uses a simplex of $n + 1$ points for n-dimensional vectors x. The algorithm first makes a simplex around the initial guess x_0 by adding 5% of each component $x_0(i)$ to x_0, and using these n vectors as elements of the simplex in addition to x_0. (It uses 0.00025 as component i if $x_0(i) = 0$.) Then the algorithm modifies the simplex repeatedly according to the following procedure.

Note The bold steps in the algorithm represent statements in the `fminsearch` iterative display.

---

1. Let $x(i)$ denote the list of points in the current simplex, $i = 1,...,n+1$.

2. Order the points in the simplex from lowest function value $f(x(1))$ to highest $f(x(n+1))$. At each step in the iteration, the current worst point $x(n+1)$ is discarded, and another point is accepted into the simplex (or, in the case of step 7 below, all n points with values above $f(x(1))$ are changed).

3. Generate the *reflected* point

   $$r = 2m - x(n+1),$$

   where

   $$m = \Sigma x(i)/n, \ i = 1...n,$$

   and calculate $f(r)$.

4. If $f(x(1)) \le f(r) < f(x(n))$, accept r and terminate this iteration. **Reflect**

5. If $f(r) < f(x(1))$, calculate the expansion point s

   $$s = m + 2(m - x(n+1)),$$

   and calculate $f(s)$.

   a. If $f(s) < f(r)$, accept s and terminate the iteration. **Expand**

   b. Otherwise, accept r and terminate the iteration. **Reflect**

6. If $f(r) \ge f(x(n))$, perform a *contraction* between m and the better of $x(n+1)$ and r:

   a. If $f(r) < f(x(n+1))$ (i.e., r is better than $x(n+1)$), calculate

      $$c = m + (r - m)/2$$

      and calculate $f(c)$. If $f(c) < f(r)$, accept c and terminate the iteration. **Contract outside** Otherwise, continue with Step 7 (Shrink).

   b. If $f(r) \ge f(x(n+1))$, calculate

      $$cc = m + (x(n+1) - m)/2$$

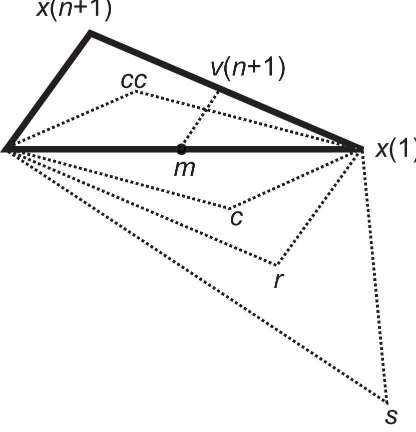
and calculate $f(cc)$. If $f(cc) < f(x(n+1))$, accept cc and terminate the iteration. **Contract inside** Otherwise, continue with Step 7 (Shrink).

7 Calculate the n points

$$v(i) = x(1) + (x(i) - x(1))/2$$

and calculate $f(v(i))$, $i = 2,...,n+1$. The simplex at the next iteration is $x(1)$, $v(2),...,v(n+1)$. **Shrink**

Here is a picture of the points that may be calculated in the procedure, along with each possible new simplex. The original simplex has a bold outline.



The iterations proceed until a stopping criterion is met.

Unconstrained Nonlinear Optimization Examples

In this section...

"Example: fminunc Unconstrained Minimization" on page 4-14

"Example: Nonlinear Minimization with Gradient and Hessian" on page 4-16

"Example: Nonlinear Minimization with Gradient and Hessian Sparsity Pattern" on page 4-17

Example: fminunc Unconstrained Minimization

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

$$\min_x f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right). \quad \textbf{(4-16)}$$

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)
```

This produces the following output:

```
Optimization terminated: relative infinity-norm
 of gradient less than options.TolFun.

x =
```

```
    0.5000   -1.0000

fval =
  3.6609e-015

exitflag =
     1

output =
   iterations: 8
    funcCount: 66
     stepsize: 1
firstorderopt: 1.2284e-007
    algorithm: 'medium-scale: Quasi-Newton line search'
      message: 'Optimization terminated: relative infinity-norm
                 of gradient less than options.TolFun.'
```

The `exitflag` tells whether the algorithm converged. `exitflag = 1` means a local minimum was found. The meanings of flags are given in function reference pages.

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`, the type of algorithm used in `algorithm`, and the exit message (the reason the algorithm stopped).

Pass the variable `options` 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. Create an `options` structure using the `optimset` function:

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

You can also create an options structure by exporting from the Optimization Tool.

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 7-7 for more options and information.

Example: 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

$$f(x) = \sum_{i=1}^{n-1} \left(\left(x_i^2\right)^{\left(x_{i+1}^2+1\right)} + \left(x_{i+1}^2\right)^{\left(x_i^2+1\right)} \right), \tag{4-17}$$

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.

The file is lengthy so is not included here. 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 about 7 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: 7
        funcCount: 8
      cgiterations: 7
    firstorderopt: 4.7948e-010
        algorithm: 'large-scale: trust-region Newton'
          message: [1x137 char]
```

Example: 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 =
  7.4739e-017
output =
       iterations: 7
        funcCount: 8
      cgiterations: 7
    firstorderopt: 7.9822e-010
        algorithm: 'large-scale: trust-region Newton'
          message: [1x137 char]
```

Constrained Nonlinear Optimization

In this section...

"Definition" on page 4-20
"fmincon Trust Region Reflective Algorithm" on page 4-20
"fmincon Active Set Algorithm" on page 4-26
"fmincon Interior Point Algorithm" on page 4-35
"fminbnd Algorithm" on page 4-39
"fseminf Problem Formulation and Algorithm" on page 4-39

Definition

Constrained minimization is the problem of finding a vector x that is a local minimum to a scalar function $f(x)$ subject to constraints on the allowable x:

$$\min_x f(x)$$

such that one or more of the following holds: $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, $l \le x \le u$. There are even more constraints used in semi-infinite programming; see "fseminf Problem Formulation and Algorithm" on page 4-39.

fmincon Trust Region Reflective Algorithm

Trust-Region Methods for Nonlinear Minimization

Many of the methods used in Optimization Toolbox solvers 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,

$$\min_s \{q(s),\ s \in N\}. \tag{4-18}$$

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

$$\min\left\{\frac{1}{2}s^T Hs + s^T g \ \text{ such that } \ \|Ds\| \le \Delta\right\}, \tag{4-19}$$

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 $\|\ .\ \|$ is the 2-norm. Good algorithms exist for solving Equation 4-19 (see [48]); 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 Equation 4-19. 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 Equation 4-19, have been proposed in the literature ([42] and [50]). The approximation approach followed in

Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). Once the subspace S has been computed, the work to solve Equation 4-19 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 solver defines S as the linear space spanned by s_1 and s_2, 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

$$H \cdot s_2 = -g, \tag{4-20}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{4-21}$$

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 sketch of unconstrained minimization using trust-region ideas is now easy to give:

1. Formulate the two-dimensional trust-region subproblem.
2. Solve Equation 4-19 to determine the trial step s.
3. If $f(x + s) < f(x)$, then $x = x + s$.
4. 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 [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat 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 Gradient Method

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.

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., $d^T Hd \leq 0$. 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 4-3.

Linear Equality Constraints

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 Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\}, \qquad \textbf{(4-22)}$$

where A is an m-by-n matrix ($m \leq n$). Some Optimization Toolbox solvers preprocess A to remove strict linear dependencies using a technique based on the LU-factorization of A^T [46]. Here A is assumed to be of rank m.

The method used to solve Equation 4-22 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 [46], 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

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \tag{4-23}$$

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., $C = H$. See [46] for more details.

Box Constraints

The box constrained problem is of the form

$$\min\{f(x) \text{ such that } l \le x \le u\}, \tag{4-24}$$

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 step size.

The scaled modified Newton step arises from examining the Kuhn-Tucker necessary conditions for Equation 4-24,

$$\left(D(x)\right)^{-2} g = 0, \tag{4-25}$$

where

$$D(x) = \operatorname{diag}\left(\left|v_k\right|^{-1/2}\right),$$

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 $g_i \geq 0$ and $l_i > -\infty$ then $v_i = x_i - l_i$
- If $g_i < 0$ and $u_i = \infty$ then $v_i = -1$
- If $g_i \geq 0$ and $l_i = -\infty$ then $v_i = 1$

The nonlinear system Equation 4-25 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 Equation 4-25 is defined as the solution to the linear system

$$\hat{M}Ds^N = -\hat{g} \tag{4-26}$$

at the kth iteration, where

$$\hat{g} = D^{-1}g = \text{diag}\left(|v|^{1/2}\right)g, \tag{4-27}$$

and

$$\hat{M} = D^{-1}HD^{-1} + \text{diag}(g)J^v. \tag{4-28}$$

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, $J^v = \text{diag}(\text{sign}(g))$. At a point where $g_i = 0$, v_i might not be differentiable.

$J_{ii}^v = 0$ 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 step size. 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 $p^R{}_i = -p_i$.

fmincon Active Set Algorithm

Introduction

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 Karush-Kuhn-Tucker (KKT) equations. The KKT 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 $G_i(x)$, $i = 1,...,m$, are convex functions, then the KKT equations are both necessary and sufficient for a global solution point.

Referring to GP (Equation 4-1), the Kuhn-Tucker equations can be stated as

$$\begin{aligned} \nabla f(x^*) + \sum_{i=1}^{m} \lambda_i \cdot \nabla G_i(x^*) = 0 \\ \lambda_i \cdot G_i(x^*) = 0, \quad i = 1,\ldots,m_e \\ \lambda_i \geq 0, \quad i = m_e + 1,\ldots,m, \end{aligned} \tag{4-29}$$

in addition to the original constraints in Equation 4-1.

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 (λ_i, $i = 1,...,m$) 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 0. This is stated implicitly in the last two Kuhn-Tucker equations.

The solution of the KKT 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 KKT 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).

The `'active-set'` algorithm is not a large-scale algorithm; see “Large-Scale vs. Medium-Scale Algorithms” on page 2-45.

Sequential Quadratic Programming (SQP)

SQP methods represent the state of the art in nonlinear programming methods. Schittkowski [36], 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 [22], and Powell ([32] and [33]), 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 [13], Gill et al. [19], Powell [35], and Schittkowski [23]. The general method, however, is stated here.

Given the problem description in GP (Equation 4-1) the principal idea is the formulation of a QP subproblem based on a quadratic approximation of the Lagrangian function.

$$L(x,\lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g_i(x). \tag{4-30}$$

Here you simplify Equation 4-1 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

$$\min_{d \in \Re^n} \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d$$
$$\nabla g_i(x_k)^T d + g_i(x_k) = 0, \quad i = 1,\ldots,m_e$$
$$\nabla g_i(x_k)^T d + g_i(x_k) \leq 0, \quad i = m_e + 1,\ldots,m. \qquad \textbf{(4-31)}$$

This subproblem can be solved using any QP algorithm (see, for instance, "Quadratic Programming Solution" on page 4-31). The solution is used to form a new iterate

$$x_{k+1} = x_k + a_k d_k.$$

The step length parameter a_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 4-29). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function (Equation 4-30). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see "Updating the Hessian Matrix" on page 4-29) 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 with an additional nonlinear inequality constraint, $g(x)$,

$$x_1^2 + x_2^2 - 1.5 \leq 0. \qquad \textbf{(4-32)}$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. SQP Method on Nonlinear Linearly Constrained Rosenbrock's Function (Eq. 3-2) on page 4-29 shows the path to the solution point $x = [0.9072,0.8228]$ starting at $x = [-1.9,2.0]$.

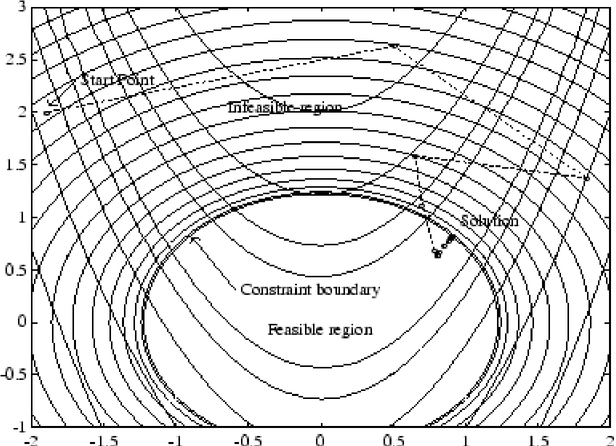


Figure 4-3: SQP Method on Nonlinear Linearly Constrained Rosenbrock's Function (Eq. 3-2)

SQP Implementation

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

- "Updating the Hessian Matrix" on page 4-29
- "Quadratic Programming Solution" on page 4-31
- "Line Search and Merit Function" on page 4-35

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

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T s_k^T s_k H_k}{s_k^T H_k s_k}, \quad \text{(4-33)}$$

where

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

Powell [33] 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 $q_k{*}s_k$ is repeatedly halved. This procedure is continued until $q_k^T s_k$ is greater than or equal to a small negative tolerance. 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,

$$q_k = q_k + wv, \tag{4-34}$$

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.

$$\begin{aligned} \min_{d \in \Re^n} q(d) &= \frac{1}{2} d^T H d + c^T d, \\ A_i d &= b_i, \quad i = 1,...,m_e \\ A_i d &\leq b_i, \quad i = m_e + 1,...,m. \end{aligned} \qquad \textbf{(4-35)}$$

The method used in Optimization Toolbox functions is an active set strategy (also known as a projection method) similar to that of Gill et al., described in [18] and [17]. 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

$$Z_k = Q[:, l+1:m], \tag{4-36}$$

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

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p. \tag{4-37}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c. \tag{4-38}$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c. \tag{4-39}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k, \text{ where } \hat{d}_k = Z_k^T p. \tag{4-40}$$

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 (Equation 4-36). 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

$$\alpha = \min_{i \in \{1,\ldots,m\}} \left\{ \frac{-(A_i x_k - b_i)}{A_i d_k} \right\}, \tag{4-41}$$

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

$$\bar{A}_k^T \lambda_k = c. \tag{4-42}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 4-36). 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

$$\begin{aligned}
&\min_{\gamma \in \Re,\ x \in \Re^n} \gamma \quad \text{such that} \\
&A_i x = b_i, \qquad i = 1,...,m_e \\
&A_i x - \gamma \le b_i, \quad i = m_e + 1,...,m.
\end{aligned} \tag{4-43}$$

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 4-43 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).

$$\hat{d}_k = -Z_k Z_k^T g_k. \tag{4-44}$$

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

$$H\hat{d}_1 = -g_k, \tag{4-45}$$

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

$$x_{k+1} = x_k + \alpha d_k. \tag{4-46}$$

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 [22] and Powell [33] of the following form is used in this implementation.

$$\Psi(x) = f(x) + \sum_{i=1}^{m_e} r_i \cdot g_i(x) + \sum_{i=m_e+1}^{m} r_i \cdot \max[0, g_i(x)]. \tag{4-47}$$

Powell recommends setting the penalty parameter

$$r_i = \left(r_{k+1}\right)_i = \max_i \left\{ \lambda_i, \frac{(r_k)_i + \lambda_i}{2} \right\}, \quad i = 1,...,m. \tag{4-48}$$

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

$$r_i = \frac{\|\nabla f(x)\|}{\|\nabla g_i(x)\|}, \tag{4-49}$$

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.

fmincon Interior Point Algorithm

Barrier Function

The interior-point approach to constrained minimization is to solve a sequence of approximate minimization problems. The original problem is

$$\min_{x} f(x), \text{ subject to } h(x) = 0 \text{ and } g(x) \le 0. \tag{4-50}$$

For each $\mu > 0$, the approximate problem is

$$\min_{x,s} f_{\mu}(x,s) = \min_{x,s} f(x) - \mu \sum_{i} \ln(s_i), \text{ subject to } h(x) = 0 \text{ and } g(x) + s = 0. \tag{4-51}$$

There are as many slack variables s_i as there are inequality constraints g. The s_i are restricted to be positive to keep $\ln(s_i)$ bounded. As μ decreases to zero, the minimum of f_{μ} should approach the minimum of f. The added logarithmic term is called a *barrier function*. This method is described in [40], [41], and [51].

The approximate problem Equation 4-51 is a sequence of equality constrained problems. These are easier to solve than the original inequality-constrained problem Equation 4-50.

To solve the approximate problem, the algorithm uses one of two main types of steps at each iteration:

- A *direct* step in (x, s). This step attempts to solve the KKT equations, Equation 2-3 and Equation 2-4, for the approximate problem via a linear approximation. This is also called a *Newton step*.
- A *CG* (conjugate gradient) step, using a trust region.

By default, the algorithm first attempts to take a direct step. If it cannot, it attempts a CG step. One case where it does not take a direct step is when the approximate problem is not locally convex near the current iterate.

At each iteration the algorithm decreases a *merit function*, such as

$$f_{\mu}(x,s) + \nu \left\| \left(h(x), g(x) + s \right) \right\|.$$

The parameter ν may increase with iteration number in order to force the solution towards feasibility. If an attempted step does not decrease the merit function, the algorithm rejects the attempted step, and attempts a new step.

Direct Step

The following variables are used in defining the direct step:

- H denotes the Hessian of the Lagrangian of f_μ:

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j \lambda_j \nabla^2 h_j(x). \tag{4-52}$$

- J_g denotes the Jacobian of the constraint function g.
- J_h denotes the Jacobian of the constraint function h.
- S = diag(s).
- λ denotes the Lagrange multiplier vector associated with constraints g
- Λ = diag(λ).
- y denotes the Lagrange multiplier vector associated with h.
- e denote the vector of ones the same size as g.

Equation 4-53 defines the direct step (Δx, Δs):

$$\begin{bmatrix} H & 0 & J_h^T & J_g^T \\ 0 & S\Lambda & 0 & -S \\ J_h & 0 & I & 0 \\ J_g & -S & 0 & I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ -\Delta y \\ -\Delta\lambda \end{bmatrix} = -\begin{bmatrix} \nabla f - J_h^T y - J_g^T \lambda \\ S\lambda - \mu e \\ h \\ g + s \end{bmatrix}. \tag{4-53}$$

This equation comes directly from attempting to solve Equation 2-3 and Equation 2-4 using a linearized Lagrangian.

In order to solve this equation for (Δx, Δs), the algorithm makes an LDL factorization of the matrix. (See Example 4 — The Structure of D in the MATLAB `ldl` function reference page.) This is the most computationally expensive step. One result of this factorization is a determination of whether the projected Hessian is positive definite or not; if not, the algorithm uses a conjugate gradient step, described in the next section.

Conjugate Gradient Step

The conjugate gradient approach to solving the approximate problem Equation 4-51 is similar to other conjugate gradient calculations. In this case, the algorithm adjusts both x and s, keeping the slacks s positive. The approach is to minimize a quadratic approximation to the approximate problem in a trust region, subject to linearized constraints.

Specifically, let R denote the radius of the trust region, and let other variables be defined as in “Direct Step” on page 4-37. The algorithm obtains Lagrange multipliers by approximately solving the KKT equations

$$\nabla_x L = \nabla_x f(x) + \sum_i \lambda_i \nabla g_i(x) + \sum_j y_j \nabla h_j(x) = 0,$$

in the least-squares sense, subject to λ being positive. Then it takes a step (Δx, Δs) to approximately solve

$$\min_{\Delta x, \Delta s} \nabla f^T \Delta x + \frac{1}{2} \Delta x^T \nabla_{xx}^2 L \Delta x + \mu e^T S^{-1} \Delta s + \frac{1}{2} \Delta s^T S^{-1} \Lambda \Delta s, \quad \textbf{(4-54)}$$

subject to the linearized constraints

$$g(x) + J_g \Delta x + \Delta s = 0, \quad h(x) + J_h \Delta x = 0. \quad \textbf{(4-55)}$$

To solve Equation 4-55, the algorithm tries to minimize a norm of the linearized constraints inside a region with radius scaled by R. Then Equation 4-54 is solved with the constraints being to match the residual from solving Equation 4-55, staying within the trust region of radius R, and keeping s strictly positive. For details of the algorithm and the derivation, see [40], [41], and [51]. For another description of conjugate gradients, see “Preconditioned Conjugate Gradient Method” on page 4-23.

Interior-Point Algorithm Options

Here are the meanings and effects of several options in the interior-point algorithm.

- `AlwaysHonorConstraints` — When set to `'bounds'`, every iterate satisfies the bound constraints you have set. When set to `'none'`, the algorithm may violate bounds during intermediate iterations.

- `Hessian` — When set to:
  - `'user-supplied'`, pass the Hessian of the Lagrangian in a user-supplied function, whose function handle must be given in the option `HessFcn`.
  - `'bfgs'`, `fmincon` calculates the Hessian by a dense quasi-Newton approximation.
  - `'lbfgs'`, `fmincon` calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation.
  - `'fin-diff-grads'`, `fmincon` calculates a Hessian-times-vector product by finite differences of the gradient(s); other options need to be set appropriately.

  You can also give a separate function for Hessian-times-vector. See "Hessian" on page 9-41 for more details on these options.
- `InitBarrierParam` — The starting value for μ. By default, this is 0.1.
- `ScaleProblem` — When set to `'obj-and-constr'`, the algorithm works with scaled versions of the objective function and constraints. It carefully scales them by their initial values. Disable scaling by setting `ScaleProblem` to `'none'`.
- `SubproblemAlgorithm` — Determines whether or not to attempt the direct Newton step. The default setting `'ldl-factorization'` allows this type of step to be attempted. The setting `'cg'` allows only conjugate gradient steps.

For a complete list of options see "Interior-Point Algorithm" on page 9-50.

fminbnd Algorithm

`fminbnd` is a solver available in any MATLAB installation. It solves for a local minimum in one dimension within a bounded interval. It is not based on derivatives. Instead, it uses golden-section search and parabolic interpolation.

fseminf Problem Formulation and Algorithm

fseminf Problem Formulation

`fseminf` addresses optimization problems with additional types of constraints compared to those addressed by `fmincon`. The formulation of `fmincon` is

$$\min_x f(x)$$

such that $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

fseminf adds the following set of semi-infinite constraints to those already given. For w_j in a one- or two-dimensional bounded interval or rectangle I_j, for a vector of continuous functions $K(x, w)$, the constraints are

$$K_j(x, w_j) \le 0 \text{ for all } w_j I_j.$$

The term "dimension" of an fseminf problem means the maximal dimension of the constraint set I: 1 if all I_j are intervals, and 2 if at least one I_j is a rectangle. The size of the vector of K does not enter into this concept of dimension.

The reason this is called semi-infinite programming is that there are a finite number of variables (x and w_j), but an infinite number of constraints. This is because the constraints on x are over a set of continuous intervals or rectangles I_j, which contains an infinite number of points, so there are an infinite number of constraints: $K_j(x, w_j) \le 0$ for an infinite number of points w_j.

You might think a problem with an infinite number of constraints is impossible to solve. fseminf addresses this by reformulating the problem to an equivalent one that has two stages: a maximization and a minimization. The semi-infinite constraints are reformulated as

$$\max_{w_j \in I_j} K_j(x, w_j) \le 0 \text{ for all } j = 1, \ldots, |K|, \tag{4-56}$$

where $|K|$ is the number of components of the vector K; i.e., the number of semi-infinite constraint functions. For fixed x, this is an ordinary maximization over bounded intervals or rectangles.

fseminf further simplifies the problem by making piecewise quadratic or cubic approximations $\kappa_j(x, w_j)$ to the functions $K_j(x, w_j)$, for each x that the solver visits. fseminf considers only the maxima of the interpolation function $\kappa_j(x, w_j)$, instead of $K_j(x, w_j)$, in Equation 4-56. This reduces the original problem, minimizing a semi-infinitely constrained function, to a problem with a finite number of constraints.

Sampling Points. Your semi-infinite constraint function must provide a set of sampling points, points used in making the quadratic or cubic approximations. To accomplish this, it should contain:

- The initial spacing `s` between sampling points w
- A way of generating the set of sampling points w from `s`

The initial spacing `s` is a $|K|$-by-2 matrix. The jth row of `s` represents the spacing for neighboring sampling points for the constraint function K_j. If K_j depends on a one-dimensional w_j, set `s(j,2) = 0`. `fseminf` updates the matrix `s` in subsequent iterations.

`fseminf` uses the matrix `s` to generate the sampling points w, which it then uses to create the approximation $\kappa_j(x, w_j)$. Your procedure for generating w from `s` should keep the same intervals or rectangles I_j during the optimization.

Example of Creating Sampling Points. Consider a problem with two semi-infinite constraints, K_1 and K_2. K_1 has one-dimensional w_1, and K_2 has two-dimensional w_2. The following code generates a sampling set from w_1 = 2 to 12:

```
% Initial sampling interval
if isnan(s(1,1))
    s(1,1) = .2;
    s(1,2) = 0;
end

% Sampling set
w1 = 2:s(1,1):12;
```

`fseminf` specifies `s` as `NaN` when it first calls your constraint function. Checking for this allows you to set the initial sampling interval.

The following code generates a sampling set from w_2 in a square, with each component going from 1 to 100, initially sampled more often in the first component than the second:

```
% Initial sampling interval
if isnan(s(1,1))
   s(2,1) = 0.2;
```

```
    s(2,2) = 0.5;
end

% Sampling set
w2x = 1:s(2,1):100;
w2y = 1:s(2,2):100;
[wx,wy] = meshgrid(w2x,w2y);
```

The preceding code snippets can be simplified as follows:

```
% Initial sampling interval
if isnan(s(1,1))
    s = [0.2 0;0.2 0.5];
end

% Sampling set
w1 = 2:s(1,1):12;
w2x = 1:s(2,1):100;
w2y = 1:s(2,2):100;
[wx,wy] = meshgrid(w2x,w2y);
```

fseminf Algorithm

`fseminf` essentially reduces the problem of semi-infinite programming to a problem addressed by `fmincon`. `fseminf` takes the following steps to solve semi-infinite programming problems:

1. At the current value of x, `fseminf` identifies all the $w_{j,i}$ such that the interpolation $\kappa_j(x, w_{j,i})$ is a local maximum. (The maximum refers to varying w for fixed x.)

2. `fseminf` takes one iteration step in the solution of the `fmincon` problem:

   $$\min_x f(x)$$

   such that $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$, where $c(x)$ is augmented with all the maxima of $\kappa_j(x, w_j)$ taken over all $w_j \, I_j$, which is equal to the maxima over j and i of $\kappa_j(x, w_{j,i})$.

3. `fseminf` checks if any stopping criterion is met at the new point x (to halt the iterations); if not, it continues to step 4.

4. `fseminf` checks if the discretization of the semi-infinite constraints needs updating, and updates the sampling points appropriately. This provides an updated approximation $\kappa_j(x, w_j)$. Then it continues at step 1.

Constrained Nonlinear Optimization Examples

In this section...

"Example: Nonlinear Inequality Constraints" on page 4-44

"Example: Bound Constraints" on page 4-46

"Example: Constraints With Gradients" on page 4-47

"Example: Constrained Minimization Using fmincon's Interior-Point Algorithm With Analytic Hessian" on page 4-50

"Example: Equality and Inequality Constraints" on page 4-57

"Example: Nonlinear Minimization with Bound Constraints and Banded Preconditioner" on page 4-58

"Example: Nonlinear Minimization with Equality Constraints" on page 4-62

"Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64

"Example: One-Dimensional Semi-Infinite Constraints" on page 4-68

"Example: Two-Dimensional Semi-Infinite Constraint" on page 4-70

Example: Nonlinear Inequality Constraints

If inequality constraints are added to Equation 4-16, the resulting problem can be solved by the `fmincon` function. For example, find x that solves

$$\min_x f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right). \tag{4-57}$$

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 $c(x) \leq 0$, you must rewrite your constraints in the form

$$\begin{aligned} &x_1 x_2 - x_1 - x_2 + 1.5 \leq 0, \\ &-x_1 x_2 - 10 \leq 0. \end{aligned} \tag{4-58}$$

Step 1: Write an M-file objfun.m for the objective function.

```
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 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 3: Invoke constrained optimization routine.

```
x0 = [-1,1];     % Make a starting guess at the solution
options = optimset('Algorithm','active-set');
[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 numbers close to zero, such as

```
c =
  1.0e-007 *
   -0.9032
    0.9032

ceq =
     []
```

Note that both constraint values are, to within a small tolerance, less than or equal to 0; that is, x satisfies $c(x) \leq 0$.

Example: Bound Constraints

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 Equation 4-57 to be greater than 0 (i.e., $x_1 \geq 0$, $x_1 \geq 0$), use the commands

```
x0 = [-1,1];        % Make a starting guess at the solution
lb = [0,0];         % Set lower bounds
ub = [ ];           % No upper bounds
options = optimset('Algorithm','active-set');
[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 $x_1 \leq 10$, $x_2 \geq 0$. 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.

Example: Constraints 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 Equation 4-57 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
if nargout  > 1
    G = [ f + exp(x(1)) * (8*x(1) + 4*x(2)),
    exp(x(1))*(4*x(1)+4*x(2)+2)];
end
```

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;
% No nonlinear equality constraints
ceq=[];
% Gradient of the constraints
if nargout > 2
    DC= [x(2)-1, -x(2);
        x(1)-1, -x(1)];
    DCeq = [];
end
```

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

$$\nabla f = \begin{bmatrix} e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right) + e^{x_1}\left(8x_1 + 4x_2\right) \\ e^{x_1}\left(4x_1 + 4x_2 + 2\right) \end{bmatrix}. \tag{4-59}$$

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

$$\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_2}{\partial x_1} \\ \frac{\partial c_1}{\partial x_2} & \frac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}. \tag{4-60}$$

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 the constrained optimization routine.

```
x0 = [-1,1];            % Starting guess
options = optimset('Algorithm','active-set');
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 =
     []
```

Example: Constrained Minimization Using fmincon's Interior-Point Algorithm With Analytic Hessian

`fmincon`'s interior-point algorithm can accept a Hessian function as an input. When you supply a Hessian, you may obtain a faster, more accurate solution to a constrained minimization problem.

The constraint set for this example is the intersection of the interior of two cones—one pointing up, and one pointing down. The constraint function `c` is a two-component vector, one component for each cone. Since this is a three-dimensional example, the gradient of the constraint `c` is a 3-by-2 matrix.

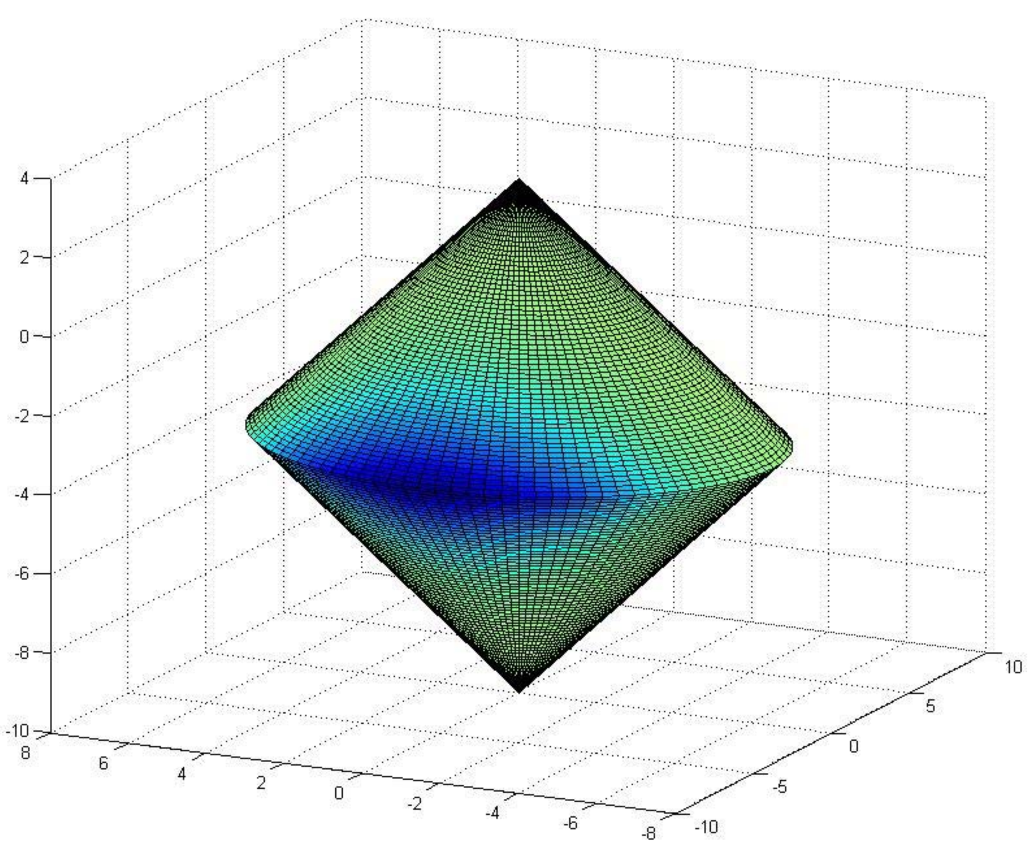
```
function [c ceq gradc gradceq] = twocone(x)
% This constraint is two cones, z > -10 + r
% and z < 3 - r

ceq = [];
r = sqrt(x(1)^2 + x(2)^2);
c = [-10+r-x(3);
    x(3)-3+r];

if nargout > 2

    gradceq = [];
    gradc = [x(1)/r,x(1)/r;
       x(2)/r,x(2)/r;
       -1,1];

end
```

The objective function grows rapidly negative as the `x(1)` coordinate becomes negative. Its gradient is a three-element vector.

```
function [f gradf] = bigtoleft(x)
% This is a simple function that grows rapidly negative
% as x(1) gets negative
%
f=10*x(1)^3+x(1)*x(2)^2+x(3)*(x(1)^2+x(2)^2);
```

```
if nargout > 1

   gradf=[30*x(1)^2+x(2)^2+2*x(3)*x(1);
        2*x(1)*x(2)+2*x(3)*x(2);
        (x(1)^2+x(2)^2)];

end
```

Here is a plot of the problem. The shading represents the value of the objective function. You can see that the objective function is minimized near `x = [-6.5,0,-3.5]`:



The Hessian of the Lagrangian is given by the equation:

$$\nabla_{xx}^2 L(x,\lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x).$$

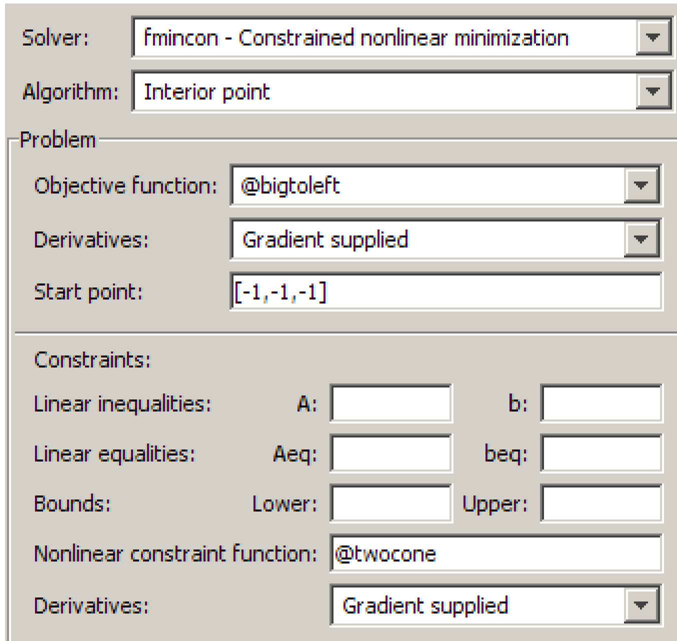
The following function computes the Hessian at a point `x` with Lagrange multiplier structure `lambda`:

```
function h = hessinterior(x,lambda)
```

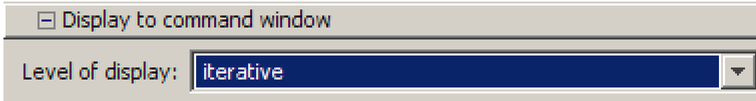
```
h = [60*x(1)+2*x(3),2*x(2),2*x(1);
    2*x(2),2*(x(1)+x(3)),2*x(2);
    2*x(1),2*x(2),0];% Hessian of f
r = sqrt(x(1)^2+x(2)^2);% radius
rinv3 = 1/r^3;
hessc = [(x(2))^2*rinv3,-x(1)*x(2)*rinv3,0;
    -x(1)*x(2)*rinv3,x(1)^2*rinv3,0;
    0,0,0];% Hessian of both c(1) and c(2)
h = h + lambda.ineqnonlin(1)*hessc + lambda.ineqnonlin(2)*hessc;
```

Run this problem using the interior-point algorithm in `fmincon`. To do this using the Optimization Tool:

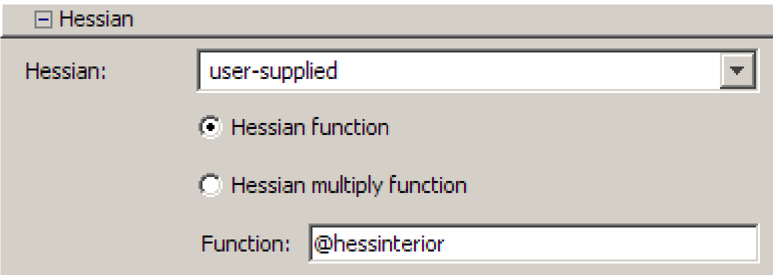
1 Set the problem as in the following figure.



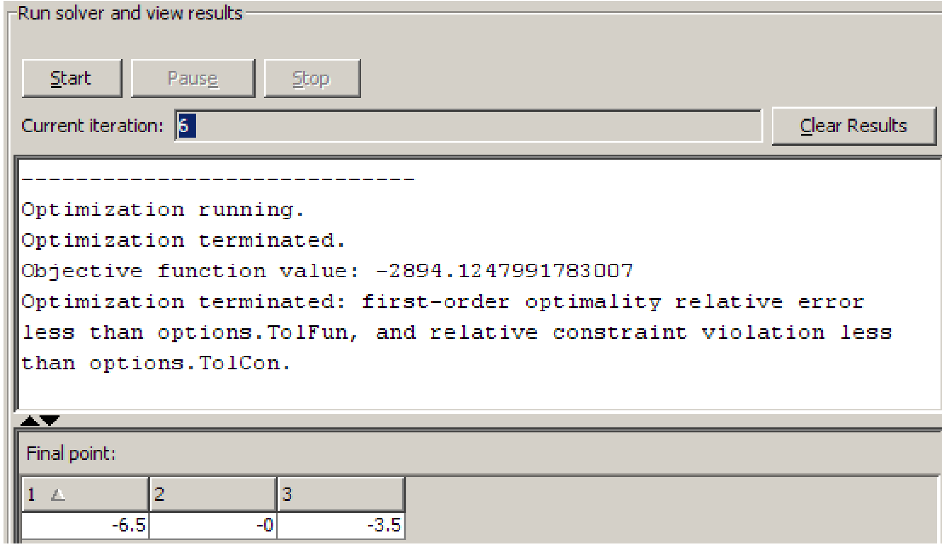
2 For iterative output, scroll to the bottom of the **Options** pane and select **Level of display**, `iterative`.



3 In the **Options** pane, give the analytic Hessian function handle.



4 Under **Run solver and view results**, click **Start**.



To perform the minimization at the command line:

1 Set `options` as follows:

```
options = optimset('Algorithm','interior-point',...
        'Display','iter','GradObj','on','GradConstr','on',...
        'Hessian','user-supplied','HessFcn',@hessinterior);
```

2 Run `fmincon` with starting point [–1,–1,–1], using the `options` structure:

```
[x fval mflag output]=fmincon(@bigtoleft,[-1,-1,-1],...
           [],[],[],[],[],[],@twocone,options)
```

The output is:

```
                                            First-order     Norm of
Iter F-count         f(x)  Feasibility   optimality        step
  0      1   -1.300000e+001   0.000e+000   3.067e+001
  1      2   -2.011543e+002   0.000e+000   1.739e+002  1.677e+000
  2      3   -1.270471e+003   9.844e-002   3.378e+002  2.410e+000
  3      4   -2.881667e+003   1.937e-002   1.079e+002  2.206e+000
  4      5   -2.931003e+003   2.798e-002   5.813e+000  6.006e-001
  5      6   -2.894085e+003   0.000e+000   2.352e-002  2.800e-002
  6      7   -2.894125e+003   0.000e+000   5.981e-005  3.048e-005
Optimization terminated: first-order optimality relative error
less than options.TolFun, and relative constraint violation less
than options.TolCon.

x =
   -6.5000   -0.0000   -3.5000

fval =
 -2.8941e+003

mflag =
     1

output =
         iterations: 6
          funcCount: 7
    constrviolation: 0
           stepsize: 3.0479e-005
          algorithm: 'interior-point'
      firstorderopt: 5.9812e-005
       cgiterations: 3
```

```
             message: [1x148 char]
```

If you do not use a Hessian function, `fmincon` takes 9 iterations to converge, instead of 6:

```
options = optimset('Algorithm','interior-point',...
       'Display','iter','GradObj','on','GradConstr','on');
[x fval mflag output]=fmincon(@bigtoleft,[-1,-1,-1],...
          [],[],[],[],[],[],@twocone,options)

                                            First-order     Norm of
Iter F-count         f(x)  Feasibility   optimality        step
  0      1   -1.300000e+001   0.000e+000   3.067e+001
  1      2   -7.259551e+003   2.495e+000   2.414e+003  8.344e+000
  2      3   -7.361301e+003   2.529e+000   2.767e+001  5.253e-002
  3      4   -2.978165e+003   9.392e-002   1.069e+003  2.462e+000
  4      8   -3.033486e+003   1.050e-001   8.282e+002  6.749e-001
  5      9   -2.893740e+003   0.000e+000   4.186e+001  1.053e-001
  6     10   -2.894074e+003   0.000e+000   2.637e-001  3.565e-004
  7     11   -2.894124e+003   0.000e+000   2.340e-001  1.680e-004
  8     12   -2.894125e+003   2.830e-008   1.180e-001  6.374e-004
  9     13   -2.894125e+003   2.939e-008   1.423e-004  6.484e-004
Optimization terminated: first-order optimality relative error
less than options.TolFun, and relative constraint violation less
than options.TolCon.

x =
   -6.5000   -0.0000   -3.5000

fval =
 -2.8941e+003

mflag =
     1

output =
         iterations: 9
          funcCount: 13
    constrviolation: 2.9391e-008
           stepsize: 6.4842e-004
```

```
       algorithm: 'interior-point'
   firstorderopt: 1.4235e-004
    cgiterations: 0
         message: [1x148 char]
```

Both runs lead to similar solutions, but the F-count and number of iterations are lower when using an analytic Hessian.

Example: Equality and Inequality Constraints

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('Algorithm','active-set');
[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 0 as desired.

Example: Nonlinear Minimization with Bound Constraints and Banded Preconditioner

The goal in this problem is to minimize the nonlinear function

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

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 $x_0 = x_{n+1} = 0$.

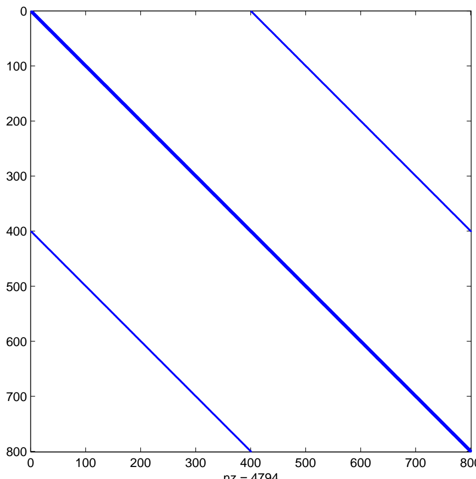
Step 1: Write an M-file tbroyfg.m that computes the objective function and the gradient of the objective

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
```

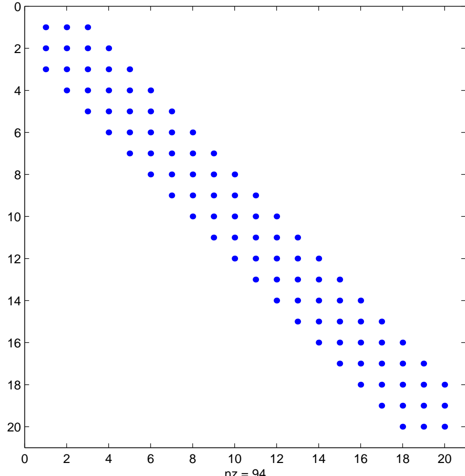
The sparsity pattern of the Hessian matrix has been predetermined and stored in the file `tbroyhstr.mat`. The sparsity structure for the Hessian of this problem is banded, as you can see in the following `spy` plot.

```
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 seven iterations, the `exitflag`, `fval`, and `output` values are

```
exitflag =
     3

fval =
  270.4790

output =
       iterations: 7
        funcCount: 8
     cgiterations: 18
    firstorderopt: 0.0163
        algorithm: 'large-scale: trust-region reflective Newton'
          message: [1x86 char]
```

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 4-24, 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 =
     3
```

```
fval =
  270.4790

output =
       iterations: 9
        funcCount: 10
     cgiterations: 15
    firstorderopt: 7.5340e-005
        algorithm: 'large-scale: trust-region reflective Newton'
          message: [1x86 char]
```

Example: Nonlinear Minimization with Equality Constraints

The trust-region reflective method for `fmincon` can handle equality constraints if no other constraints exist. Suppose you want to minimize the same objective as in Equation 4-17, 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.

The file is lengthy so is not included here. 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');
[x,fval,exitflag,output] = ...
   fmincon(fun,xstart,[],[],Aeq,beq,[],[],[],options);
```

The option `PrecondBandWidth` is `inf` by default. It means a sparse direct solver is used, instead of preconditioned conjugate gradients.

The `exitflag` value of `3` indicates that the algorithm terminated because the change in the objective function value was less than the specified tolerance `TolFun`. The final function value is given by `fval`.

```
exitflag =
     3

fval =
  205.9313

output =
       iterations: 16
        funcCount: 17
      cgiterations: 20
    firstorderopt: 7.3575e-005
        algorithm: 'large-scale: projected trust-region Newton'
          message: [1x86 char]
```

The linear equalities are satisfied at `x`.

```
norm(Aeq*x-beq)

ans =
  1.1885e-012
```

Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints

The `fmincon` interior-point and trust-region reflective algorithms, and the `fminunc` large-scale algorithm 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, 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 description applies to the trust-region reflective algorithm; the `fminunc` large-scale algorithm is similar. For the interior-point algorithm, see the `'HessMult'` option in "Hessian" on page 9-41. The objective function has the structure

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

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 optimdemos directory 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 optim directory. 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 1984-2006 The MathWorks, Inc.
%   $Revision: 1.1.6.4.2.1 $  $Date: 2008/08/11 17:13:06 $

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,x] = 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
   20      -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)

ans =

  1.9854e-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.

Example: One-Dimensional Semi-Infinite Constraints

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

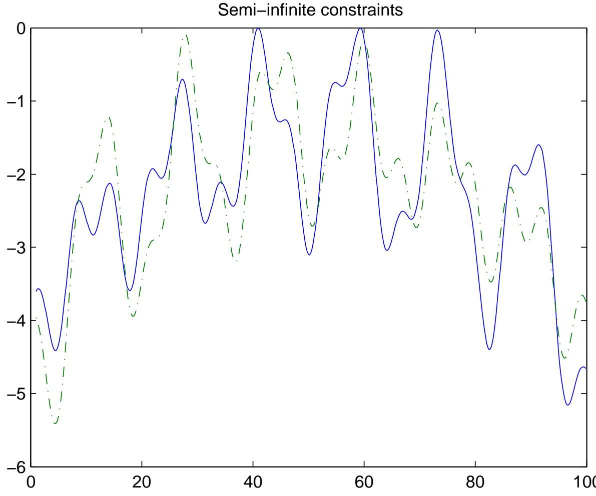
```
x =
     0.6673
     0.3013
     0.4023
```

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); % 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.

Example: Two-Dimensional Semi-Infinite Constraint

Find values of x that minimize

$$f(x) = (x_1 - 0.2)^2 + (x_2 - 0.2)^2 + (x_3 - 0.2)^2,$$

where

$$K_1(x,w) = \sin(w_1 x_1)\cos(w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \ldots$$
$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \leq 1.5,$$

for all values of w_1 and w_2 over the ranges

$1 \leq w_1 \leq 100,$
$1 \leq w_2 \leq 100,$

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
```

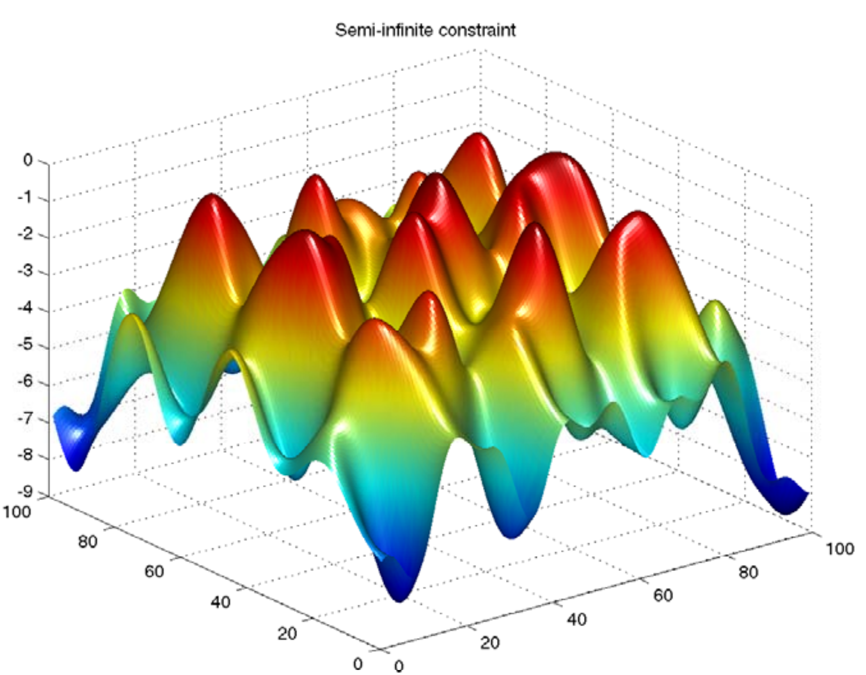
and the function value at the solution is

```
fval =
     0.0091
```

The goal was to minimize the objective $f(x)$ such that the semi-infinite constraint satisfied $K_1(x,w) \leq 1.5$. 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`.


Semi-infinite constraint
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
100
80
60
40
20
0
0
20
40
60
80
100

Linear Programming

In this section...

"Definition" on page 4-74

"Large Scale Linear Programming" on page 4-74

"Active-Set Medium-Scale linprog Algorithm" on page 4-78

"Medium-Scale linprog Simplex Algorithm" on page 4-82

Definition

Linear programming is the problem of finding a vector x that minimizes a linear function $f^T x$ subject to linear constraints:

$$\min_x f^T x$$

such that one or more of the following hold: $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u.$

Large Scale Linear Programming

Introduction

The default large-scale method is based on LIPSOL ([52]), which is a variant of Mehrotra's predictor-corrector algorithm ([47]), a primal-dual interior-point method.

Main Algorithm

The algorithm begins by applying a series of preprocessing steps (see "Preprocessing" on page 4-77). After preprocessing, the problem has the form

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \le x \le u. \end{cases} \quad \textbf{(4-61)}$$

The upper bounds constraints are implicitly included in the constraint matrix A. With the addition of primal slack variables s, Equation 4-61 becomes

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ x + s = u \\ x \geq 0,\ s \geq 0. \end{cases} \quad \textbf{(4-62)}$$

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

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \geq 0,\ w \geq 0, \end{cases} \quad \textbf{(4-63)}$$

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 Equation 4-62 and the dual Equation 4-63, are

$$F(x,y,z,s,w) = \begin{pmatrix} A \cdot x - b \\ x + s - u \\ A^T \cdot y - w + z - f \\ x_i z_i \\ s_i w_i \end{pmatrix} = 0, \\ x \geq 0,\ z \geq 0,\ s \geq 0,\ w \geq 0, \quad \textbf{(4-64)}$$

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 Equation 4-64, 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 Equation 4-62.)

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

$$\Delta v_p = -\left(F^T(v)\right)^{-1} F(v),$$

which is the Newton direction; then the so-called *corrector* direction

$$\Delta v_c = -\left(F^T(v)\right)^{-1} F\left(v + \Delta v_p\right) - \mu e,$$

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 $a > 0$ and update v to obtain the new iterate v^+:

$$v^+ = v + \alpha\left(\Delta v_p + \Delta v_c\right),$$

where the step length parameter a 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. If that solution is not adequate (the residual is too large), it performs an LDL factorization of an augmented form of the step equations to find a solution. (See Example 4 — The Structure of D in the MATLAB `ldl` function reference page.)

The algorithm then repeats these steps until the iterates converge. The main stopping criteria is a standard one:

$$\frac{\|r_b\|}{\max(1,\|b\|)} + \frac{\|r_f\|}{\max(1,\|f\|)} + \frac{\|r_u\|}{\max(1,\|u\|)} + \frac{\left|f^T x - b^T y + u^T w\right|}{\max\left(1,\left|f^T x\right|,\left|b^T y - u^T w\right|\right)} \le tol,$$

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 Equation 4-64.

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.

Active-Set Medium-Scale linprog Algorithm

The medium-scale active-set linear programming algorithm is a variant of the sequential quadratic programming algorithm used by fmincon ("Sequential Quadratic Programming (SQP)" on page 4-27). The difference is that the quadratic term is set to zero.

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.

$$\begin{aligned} &\min_{d \in \Re^n} q(d) = c^T d, \\ &\qquad A_i d = b_i, \quad i = 1,\ldots,m_e \\ &\qquad A_i d \le b_i, \quad i = m_e + 1,\ldots,m. \end{aligned}$$

The method used in Optimization Toolbox functions is an active set strategy (also known as a projection method) similar to that of Gill et al., described in [18] and [17]. 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

$$Z_k = Q[:, l+1 : m], \tag{4-65}$$

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

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p. \tag{4-66}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c. \tag{4-67}$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c. \tag{4-68}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k, \quad \text{where } \hat{d}_k = Z_k^T p. \tag{4-69}$$

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 $\overline{A}_k$. If such a step can be taken, without violation of the constraints, then this is the solution to QP (Equation 4-36). 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

$$\alpha = \min_{i \in \{1,\ldots,m\}} \left\{ \frac{-(A_i x_k - b_i)}{A_i d_k} \right\}, \tag{4-70}$$

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

$$\bar{A}_k^T \lambda_k = c. \quad \textbf{(4-71)}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 4-36). 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

$$\begin{aligned} &\min_{\gamma \in \Re,\, x \in \Re^n} \gamma \quad \text{such that} \\ &A_i x = b_i, \qquad i = 1,\ldots,m_e \\ &A_i x - \gamma \le b_i, \quad i = m_e + 1,\ldots,m. \end{aligned} \quad \textbf{(4-72)}$$

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 4-72 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).

$$\hat{d}_k = -Z_k Z_k^T g_k. \quad \textbf{(4-73)}$$

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

$$H\hat{d}_1 = -g_k, \quad \textbf{(4-74)}$$

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.

Medium-Scale linprog 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

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

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^Tx, 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 4-82
- "Preprocessing" on page 4-84
- "Using the Simplex Algorithm" on page 4-84
- "Basic and Nonbasic Variables" on page 4-85

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 4-85 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

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

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

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

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 4-85 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 4-77. 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 to 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:

$$\min_{x} f^{T} x \text{ such that } \begin{cases} A \cdot x = b, \\ lb \leq x \leq ub. \end{cases}$$

(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 $\left\{a_{i_1}, a_{i_2}, ..., a_{i_m}\right\}$ 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*.

Linear Programming Examples

In this section...

"Example: Linear Programming with Equalities and Inequalities" on page 4-86

"Example: Linear Programming with Dense Columns in the Equalities" on page 4-87

Example: Linear Programming with Equalities and Inequalities

The problem is

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ x \geq 0. \end{cases}$$

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 3.18e-015 3.62e+003 9.90e-001
```

```
  Iter    2:  8.32e-013 1.96e-015 4.32e+002 9.48e-001
  Iter    3:  3.47e-012 7.49e-015 7.78e+001 6.88e-001
  Iter    4:  5.66e-011 1.16e-015 2.38e+001 2.69e-001
  Iter    5:  1.13e-010 3.67e-015 5.05e+000 6.89e-002
  Iter    6:  5.03e-011 1.21e-016 1.64e-001 2.34e-003
  Iter    7:  5.75e-012 1.12e-016 1.09e-005 1.55e-007
  Iter    8:  8.08e-014 5.67e-013 1.09e-011 3.82e-012
Optimization terminated.
```

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
       algorithm: 'large-scale: interior point'
    cgiterations: 0
         message: 'Optimization terminated.'
```

Example: Linear Programming with Dense Columns in the Equalities

The problem is

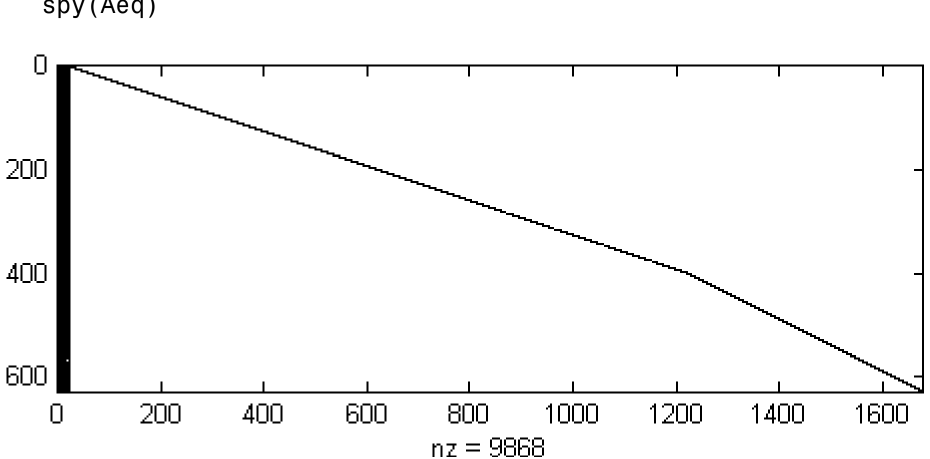
$$\min_x f^T x \text{ such that } \begin{cases} Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

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:  6.96e-007 2.00e-012 5.64e-007 1.48e+001 1.62e-003
  Iter   12:  9.35e-007 6.98e-013 3.18e-008 8.32e-001 9.09e-005
  Iter   13:  1.14e-007 2.03e-012 3.86e-009 7.26e-002 7.94e-006
  Iter   14:  1.92e-010 1.16e-012 6.55e-012 1.11e-003 1.21e-007
  Iter   15:  1.05e-013 2.50e-012 3.71e-013 8.62e-008 9.42e-012
Optimization terminated.
```

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

```
exitflag =
     1
fval =
  9.1464e+003
output =
      iterations: 15
       algorithm: 'large-scale: interior point'
    cgiterations: 0
         message: 'Optimization terminated.'
```

This time the algorithm detects dense columns in `Aeq` are detected. Therefore, 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, the algorithm uses a solution process based on LDL factorization. See “Main Algorithm” on page 4-74.

Quadratic Programming

In this section...

"Definition" on page 4-90
"Large-Scale quadprog Algorithm" on page 4-90
"Medium-Scale quadprog Algorithm" on page 4-95

Definition

Quadratic programming is the problem of finding a vector x that minimizes a quadratic function, possibly subject to linear constraints:

$$\min_x \frac{1}{2} x^T H x + c^T x$$

such that $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad l \le x \le u.$

Large-Scale quadprog Algorithm

Many of the methods used in Optimization Toolbox solvers 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,

$$\min_s \{ q(s),\ s \in N \}. \tag{4-75}$$

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

$$\min\left\{\frac{1}{2}s^T Hs + s^T g \ \text{ such that } \ \|Ds\| \le \Delta\right\}, \tag{4-76}$$

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 $\|\ .\ \|$ is the 2-norm. Good algorithms exist for solving Equation 4-76 (see [48]); 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 Equation 4-76. 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 Equation 4-76, have been proposed in the literature ([42] and [50]). The approximation approach followed in Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). Once the subspace S has been computed, the work to solve Equation 4-76 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 solver defines

S as the linear space spanned by s_1 and s_2, 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

$$H \cdot s_2 = -g, \tag{4-77}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{4-78}$$

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 sketch of unconstrained minimization using trust-region ideas is now easy to give:

1. Formulate the two-dimensional trust-region subproblem.
2. Solve Equation 4-76 to determine the trial step s.
3. If $f(x + s) < f(x)$, then $x = x + s$.
4. 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 [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat 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.

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 [45] for details of the line search.

Preconditioned Conjugate Gradient Method

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.

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 4-3.

Linear Equality Constraints

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 Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\}, \qquad \textbf{(4-79)}$$

where A is an m-by-n matrix ($m \leq n$). Some Optimization Toolbox solvers preprocess A to remove strict linear dependencies using a technique based on the LU-factorization of A^T [46]. Here A is assumed to be of rank m.

The method used to solve Equation 4-79 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 [46], 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

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \tag{4-80}$$

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., $C = H$. See [46] for more details.

Box Constraints

The box constrained problem is of the form

$$\min\{f(x) \text{ such that } l \le x \le u\}, \tag{4-81}$$

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 step size.

The scaled modified Newton step arises from examining the Kuhn-Tucker necessary conditions for Equation 4-81,

$$(D(x))^{-2} g = 0, \tag{4-82}$$

where

$$D(x) = \text{diag}\left(\left|v_k\right|^{-1/2}\right),$$

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 Equation 4-82 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 Equation 4-82 is defined as the solution to the linear system

$$\hat{M}Ds^N = -\hat{g} \tag{4-83}$$

at the kth iteration, where

$$\hat{g} = D^{-1}g = \text{diag}\left(|v|^{1/2}\right)g, \tag{4-84}$$

and

$$\hat{M} = D^{-1}HD^{-1} + \text{diag}(g)J^v. \tag{4-85}$$

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, $J^v = \text{diag}(\text{sign}(g))$. At a point where $g_i = 0$, v_i might not be differentiable.

$J_{ii}^v = 0$ 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 step size. 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 $p^R{}_i = -p_i$.

Medium-Scale quadprog Algorithm

Recall the problem `quadprog` addresses:

$$\min_{x} \frac{1}{2} x^T H x + c^T x \qquad \text{(4-86)}$$

such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, and $l \leq x \leq u$. m is the total number of linear constraints, the sum of number of rows of A and of Aeq.

The medium-scale `quadprog` algorithm is an active-set strategy (also known as a projection method) similar to that of Gill et al., described in [18] and [17]. 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.

Active Set Iterations

In this method an active set matrix, S_k, is maintained that is an estimate of the active constraints (i.e., those that are on the constraint boundaries) at the solution point. Specifically, the active set S_k consists of the rows of Aeq, and a subset of the rows of A. S_k is updated at each iteration k, and is used to form a basis for a search direction d_k. Equality constraints always remain in the active set S_k. The search direction d_k is calculated and minimizes the objective function while remaining on active constraint boundaries. The feasible subspace for d_k is formed from a basis Z_k whose columns are orthogonal to the estimate of the active set S_k (i.e., $S_k Z_k = 0$). 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 S_k^T, where l is the number of active constraints and $l < m$. That is, Z_k is given by

$$Z_k = Q[:, l+1 : m], \qquad \text{(4-87)}$$

where

$$Q^T S_k^T = \begin{bmatrix} R \\ 0 \end{bmatrix}.$$

Once Z_k is found, a search direction d_k is sought that minimizes the objective function at d_k, where d_k is in the null space of the active constraints. That is, d_k is a linear combination of the columns of Z_k: $d_k = Z_k p$ for some vector p.

Then if you view the quadratic objective function as a function of p, by substituting for d_k, the result is

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p. \quad \textbf{(4-88)}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c. \quad \textbf{(4-89)}$$

$\nabla q(p)$ 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, 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

$$Z_k^T H Z_k p = -Z_k^T c. \quad \textbf{(4-90)}$$

The next step is

$$x_{k+1} = x_k + \alpha d_k, \;\; \text{where } d_k = Z_k^T p. \quad \textbf{(4-91)}$$

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 d_k is the exact step to the minimum of the function restricted to the null space of S_k. If such a step can be taken, without violation of the constraints, then this is the solution to QP (Equation 4-86). Otherwise, the step along 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 d_k is given by

$$\alpha = \min_{i \in \{1,\ldots,m\}} \left\{ \frac{-(A_i x_k - b_i)}{A_i d_k} \right\}, \quad \textbf{(4-92)}$$

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

Lagrange multipliers, λ_k, are calculated that satisfy the nonsingular set of linear equations

$$S_k^T \lambda_k = c. \quad \textbf{(4-93)}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 4-86). 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 initial point is not feasible, then you can find a feasible point by solving the linear programming problem

$$\begin{aligned} &\min_{\gamma \in \Re,\ x \in \Re^n} \gamma \quad \text{such that} \\ &A_i x = b_i, \qquad i = 1,\ldots,m_e \ \text{(the rows of } Aeq\text{)} \\ &A_i x - \gamma \le b_i, \quad i = m_e + 1,\ldots,m \ \text{(the rows of } A\text{).} \end{aligned} \quad \textbf{(4-94)}$$

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 4-94 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, 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 d 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):

$$d = -Z_k Z_k^T g_k. \tag{4-95}$$

If a feasible point is found using the preceding LP method, the main QP phase is entered. The search direction d_k is initialized with a search direction d_1 found from solving the set of linear equations

$$Hd_1 = -g_k, \tag{4-96}$$

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

Quadratic Programming Examples

In this section...

"Example: Quadratic Minimization with Bound Constraints" on page 4-100

"Example: Quadratic Minimization with a Dense but Structured Hessian" on page 4-102

Example: 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, and 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 =
     3
output =
       iterations: 19
        algorithm: 'large-scale: reflective trust-region'
```

```
      firstorderopt: 1.0761e-005
       cgiterations: 1640
            message: [1x206 char]
```

you can see that while convergence occurred in 19 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 =
     3
output =
       iterations: 36
        algorithm: 'large-scale: reflective trust-region'
    firstorderopt: 2.3821e-005
     cgiterations: 1547
          message: [1x206 char]
```

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 =
     3
output =
       iterations: 10
        algorithm: 'large-scale: reflective trust-region'
```

```
     firstorderopt: 1.0366e-006
      cgiterations: 0
           message: [1x206 char]
```

Using a direct solver 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.

Example: 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 4-105).

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

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

Now, define a function `runqpbox4` 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.

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 Hinfo argument. Also, B will be
```

```
% used in computing a preconditioner for PCG.
[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.
```

```
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.

Binary Integer Programming

In this section...

"Definition" on page 4-108
"bintprog Algorithm" on page 4-108

Definition

Binary integer programming is the problem of finding a binary vector x that minimizes a linear function $f^T x$ subject to linear constraints:

$$\min_x f^T x$$

such that $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad x$ binary.

bintprog 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

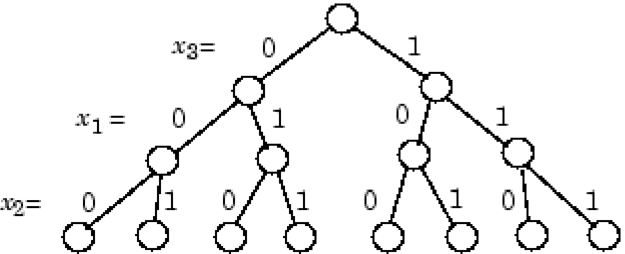
- Searches for a binary integer feasible solution
- Updates the best binary integer feasible point found so far as the search tree grows
- Verifies that no better integer feasible solution is possible by solving a series of linear programming problems

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 9-5 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 9-5 for more information.

Binary Integer Programming Example

Example: Investments with Constraints

This example uses `bintprog` to solve an integer programming problem that is not obviously a binary integer programming problem. This is done by representing each nonbinary integer-valued variable as an appropriate sum of binary variables, and by using linear constraints carefully. While the example is not particularly realistic, it demonstrates a variety of techniques:

- How to formulate nonbinary integer programming problems
- How to formulate an objective and constraints
- How to use indicator variables (y_i in the example)

Problem Statement

There are five investment opportunities labeled **1**, **2**, **3**, **4**, and **5**. The investments have the costs and payoffs listed in the following table.

Investment	Buy-In Cost	Cost/Unit	Payoff/Unit	Max # Units
1	$25	$5	$15	5
2	$35	$7	$25	4
3	$28	$6	$17	5
4	$20	$4	$13	7
5	$40	$8	$18	3

- The maximum total investment is $125.
- The problem is to maximize profit, which is payoff minus cost.
- The payoff is the sum of the units bought times the payoff/unit.
- The cost is the buy-in cost plus the cost/unit times the number of units if you buy at least one unit; otherwise, it is 0.

It is convenient to formulate this problem using the indicator variables y_i. Define these as $y_i = 1$ when corresponding quantity variable x_i is positive, and $y_i = 0$ when $x_i = 0$:

- x_i = # units purchased of investment i
- $y_i = 1$ if $x_i > 0$, $y_i = 0$ otherwise
- cost = Σ(Buy-in cost)$_i \cdot y_i$ + Σ(cost/unit)$_i \cdot x_i$
- payoff = Σ(payoff/unit)$_i \cdot x_i$
- profit = cost – payoff

In addition, there are several constraints on the investments:

- You may not invest in both **2** and **5**.
- You may invest in **1** only if you invest in at least one of **2** and **3**.
- You must invest in at least two of **3**, **4**, and **5**.
- You may not invest more than the listed maximum number of units in each investment.

The constraints are represented in symbols as follows:

- $y_2 + y_5 \le 1$
- $y_1 \le y_2 + y_3$
- $y_3 + y_4 + y_5 \ge 2$
- $x_1 \le 5$; $x_2 \le 4$; $x_3 \le 5$; $x_4 \le 7$; $x_5 \le 3$
- cost ≤ 125

bintprog Formulation

To frame this problem as a binary integer programming problem, perform the following steps:

1. Represent each integer variable x_i by three binary integer variables $z_{i,j}$, $j = 1,...,3$, as follows:

   $$x_i = z_{i,1} + 2z_{i,2} + 4z_{i,3},\ i = 1,...,5.$$

   Three $z_{i,j}$ suffice to represent x_i, since each x_i is assumed to be 7 or less. And, since $x_5 \le 3$, $z_{5,3} = 0$.

2 Combine the variables y and z into a single vector t as follows:

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}
y_1	y_2	y_3	y_4	y_5	$z_{1,1}$	$z_{1,2}$	$z_{1,3}$	$z_{2,1}$	$z_{2,2}$	$z_{2,3}$	$z_{3,1}$	$z_{3,2}$	$z_{3,3}$	$z_{4,1}$	$z_{4,2}$	$z_{4,3}$	$z_{5,1}$	$z_{5,2}$

3 Include the constraints $y_i = 0$ if and only if all the corresponding $z_{i,j} = 0$ as follows:

- $y_i \le z_{i,1} + z_{i,2} + z_{i,3}$
- $z_{i,1} + 2z_{i,2} + 4z_{i,3} \le y_i$*(Max # units$_i$)

These two inequalities enforce $y_i = 0$ if and only if all the $z_{i,j} = 0$, and they also enforce the maximum # units constraints.

4 As described in "Maximizing an Objective" on page 2-9, you find a maximum of an objective function by minimizing the negative of the objective function. So, to find the maximum profit, minimize the negative of the profit. The vector `f` that gives the negative of the profit in the form `f'*t` is

```
f = [25,35,28,20,40,-10,-20,-40,-18,-36,-72,-11,-22,-44,...
    -9,-18,-36,-10,-20]';
```

- The first five entries in `f` represent the buy-in costs; these are incurred if the corresponding $y_i = 1$.
- The next three entries, `f(6)`, `f(7)`, and `f(8)`, represent the negative of payoff minus cost per unit for investment **1**, –($15 – $5), multiplied by 1, 2, and 4 respectively.
- The entries `f(9)`, `f(10)`, and `f(11)` represent the corresponding quantities for investment **2**: –($25 – $7), multiplied by 1, 2, and 4.
- `f(12)`, `f(13)`, and `f(14)` correspond to investment **3**
- `f(15)`, `f(16)`, and `f(17)` correspond to investment **4**
- `f(18)` and `f(19)` correspond to investment **5**

5 Formulate all the constraints as inequalities of the form $A \cdot t \le b$, as required in the `bintprog` formulation "Definition" on page 4-108.

The following matrix `A` represents the constraints, along with the vector `b`:

```
A = zeros(14,19);
A(1,1:19) = [25 35 28 20 40 5 10 20 7 14 28 ...
    6 12 24 4 8 16 8 16];
A(2,1) = 1; A(2,6) = -1; A(2,7) = -1; A(2,8) = -1;
A(3,2) = 1; A(3,9) = -1; A(3,10) = -1; A(3,11) = -1;
A(4,3) = 1; A(4,12) = -1; A(4,13) = -1; A(4,14) = -1;
A(5,4) = 1; A(5,15) = -1; A(5,16) = -1; A(5,17) = -1;
A(6,5) = 1; A(6,18) = -1; A(6,19) = -1;
A(7,1) = -5; A(7,6) = 1; A(7,7) = 2; A(7,8) = 4;
A(8,2) = -4; A(8,9) = 1; A(8,10) = 2; A(8,11) = 4;
A(9,3) = -5; A(9,12) = 1; A(9,13) = 2; A(9,14) = 4;
A(10,4) = -7; A(10,15) = 1; A(10,16) = 2; A(10,17) = 4;
A(11,5) = -3; A(11,18) = 1; A(11,19) = 2;
A(12,2) = 1; A(12,5) = 1;
A(13,1) = 1; A(13,2) = -1; A(13,3) = -1;
A(14,3) = -1; A(14,4) = -1; A(14,5) = -1;
b = [125 0 0 0 0 0 0 0 0 0 0 1 0 -2]';
```

- The first row of A represents the cost structure; no more than $125 is available.
- Rows 2 through 6 represent $y_i \leq \Sigma_j z_{i,j}$, $i = 1,...,5$.
- Rows 7 through 11 represent the maximum # units constraints. They also enforce $y_i = 1$ when $\Sigma_j z_{i,j} > 0$.
- Rows 12, 13, and 14 represent the other constraints on investments.

bintprog Solution

`bintprog` solves the optimization problem as follows:

```
[t fval exitflag output] = bintprog(f,A,b);
Optimization terminated.
```

To examine the result, enter

```
t',fval

ans =
  Columns 1 through 10
     0     0     1     1     0     0     0     0     0     0
```

```
  Columns 11 through 19
     0     1     0     1     1     1     1     0     0

fval =
   -70
```

You can easily see that the only positive values of y are y_3 and y_4. The values of x that correspond to these, x_3 and x_4, are

```
t(12) + 2*t(13) + 4*t(14)
ans =
     5

t(15) + 2*t(16) + 4*t(17)
ans =
     7
```

In other words, to obtain the maximum profit, $70, invest in 5 units of **3** and 7 units of **4**. By the way, this uses only

```
28 + (5*6) + 20 + (7*4) = 106
```

of the $125 you had to invest, so there is $19 left uninvested. You can also see this by checking the first constraint, `[A*t](1)`:

```
A(1,:)*t
ans =
   106
```

Least Squares (Model Fitting)

In this section...
"Definition" on page 4-116
"Large-Scale Least Squares" on page 4-117
"Levenberg-Marquardt Method" on page 4-121
"Gauss-Newton Method" on page 4-122

Definition

Least squares, in general, is the problem of finding a vector x that is a local minimizer to a function that is a sum of squares, possibly subject to some constraints:

$$\min_x \|F(x)\|_2^2 = \min_x \sum_i F_i^2(x)$$

such that $A \cdot x \le b, \quad Aeq \cdot x = beq, \quad lb \le x \le ub.$

There are several Optimization Toolbox solvers available for various types of $F(x)$ and various types of constraints:

Solver	$F(x)$	Constraints
`\`	$C \cdot x - d$	None
`lsqnonneg`	$C \cdot x - d$	$x \ge 0$
`lsqlin`	$C \cdot x - d$	Bound, linear
`lsqnonlin`	General $F(x)$	Bound
`lsqcurvefit`	$F(x, xdata) - ydata$	Bound

There are five least-squares algorithms in Optimization Toolbox solvers, in addition to the algorithms used in `\`:

- Trust-region-reflective
- Levenberg-Marquardt

- Gauss-Newton
- `lsqlin` medium-scale (the large-scale algorithm is trust-region reflective)
- The algorithm used by `lsqnonneg`

The trust-region reflective algorithm, `lsqnonneg` algorithm, and Levenberg-Marquardt algorithm are large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-45. The Gauss-Newton and medium-scale `lsqlin` algorithms are not large-scale. For a general survey of nonlinear least-squares methods, see Dennis [8]. Specific details on the Levenberg-Marquardt method can be found in Moré [28].

Large-Scale Least Squares

Large Scale Trust-Region Reflective Least Squares

Many of the methods used in Optimization Toolbox solvers 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,

$$\min_{s}\{q(s),\ s \in N\}. \qquad \textbf{(4-97)}$$

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

$$\min\left\{\frac{1}{2}s^T Hs + s^T g \text{ such that } \|Ds\| \leq \Delta\right\}, \tag{4-98}$$

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 $\| \,.\, \|$ is the 2-norm. Good algorithms exist for solving Equation 4-98 (see [48]); 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 Equation 4-98. 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 Equation 4-98, have been proposed in the literature ([42] and [50]). The approximation approach followed in Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). Once the subspace S has been computed, the work to solve Equation 4-98 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 solver defines S as the linear space spanned by s_1 and s_2, 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

$$H \cdot s_2 = -g, \tag{4-99}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{4-100}$$

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 sketch of unconstrained minimization using trust-region ideas is now easy to give:

1. Formulate the two-dimensional trust-region subproblem.

2. Solve Equation 4-98 to determine the trial step s.

3. If $f(x + s) < f(x)$, then $x = x + s$.

4. 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 [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat 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.

Large Scale Nonlinear Least Squares

An important special case for $f(x)$ is the nonlinear least-squares problem

$$\min_x \sum_i f_i^2(x) = \min_x \|F(x)\|_2^2, \tag{4-101}$$

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 4-3. 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

$$\min \|Js + F\|_2^2, \tag{4-102}$$

(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.

Large Scale Linear Least Squares

In this case the function $f(x)$ to be solved is

$$f(x) = \|Cx + d\|_2^2,$$

possibly subject to linear constraints. 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 [45] 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.

Jacobian Multiply Function. `lsqlin` can solve the linearly-constrained least-squares problem without using the matrix C explicitly. Instead, it uses a Jacobian multiply function `jmfun`,

```
W = jmfun(Jinfo,Y,flag)
```

that you provide. The function must calculate the following products for a matrix Y:

- If `flag == 0` then `W = C'*(C*Y)`.
- If `flag > 0` then `W = C*Y`.
- If `flag < 0` then `W = C'*Y`.

This can be useful if C is large, but contains enough structure that you can write `jmfun` without forming C explicitly. For an example, see “Example: Jacobian Multiply Function with Linear Least Squares” on page 4-134.

Levenberg-Marquardt Method

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left(J(x_k)^T J(x_k) + \lambda_k I\right) d_k = -J(x_k)^T F(x_k), \quad \textbf{(4-103)}$$

or, optionally, of the equations

$$\left(J(x_k)^T J(x_k) + \lambda_k diag\left(J(x_k)^T J(x_k)\right)\right) d_k = -J(x_k)^T F(x_k), \quad \textbf{(4-104)}$$

where the scalar λ_k controls both the magnitude and direction of d_k. Set option `ScaleProblem` to `'none'` to choose Equation 4-103, and set `ScaleProblem` to `'Jacobian'` to choose Equation 4-104.

When λ_k is zero, the direction d_k is identical to that of the Gauss-Newton method. As λ_k tends to infinity, d_k tends towards the steepest descent direction, with magnitude tending to zero. 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 direction. This is illustrated in Figure 4-4, Levenberg-Marquardt Method on Rosenbrock's Function. The solution for Rosenbrock's function 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.

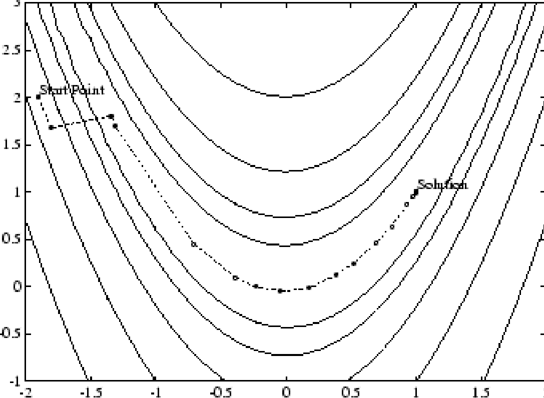


Figure 4-4: Levenberg-Marquardt Method on Rosenbrock's Function

Gauss-Newton Method

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.

$$\min_{x} f(x) = \|F(x)\|_2^2 = \sum_i F_i^2(x). \tag{4-105}$$

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, $\varphi(t)$, for vector x and scalar t. This problem can be expressed as

$$\min_{x \in \Re^n} \int_{t_1}^{t_2} (y(x,t) - \varphi(t))^2 dt, \tag{4-106}$$

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

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

$$\min_{x \in \Re^n} f(x) = \sum_{i=1}^{m} (\bar{y}(x,t_i) - \bar{\varphi}(t_i))^2, \tag{4-107}$$

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

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

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 can be minimized using a general unconstrained minimization technique, as described in “Basics of Unconstrained Optimization” on page 4-6, 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 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

$$G(x) = 2J(x)^T F(x)$$
$$H(x) = 2J(x)^T J(x) + 2Q(x), \quad \textbf{(4-108)}$$

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.

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:

$$\min_{x \in \Re^n} \|J(x_k) - F(x_k)\|_2^2. \quad \textbf{(4-109)}$$

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. Gauss-Newton Method on Rosenbrock's Function on page 4-125 shows the path to the minimum on Rosenbrock's function 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.

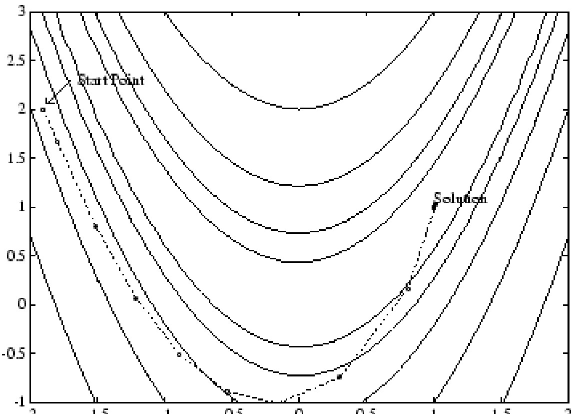


Figure 4-5: Gauss-Newton Method on Rosenbrock's Function

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

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.

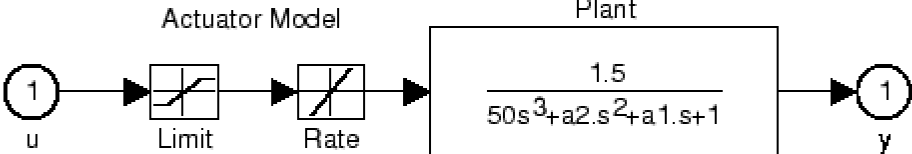
Least Squares (Model Fitting) Examples

In this section...

"Example: Using lsqnonlin With a Simulink Model" on page 4-126
"Example: Nonlinear Least-Squares with Full Jacobian Sparsity Pattern" on page 4-131
"Example: Linear Least-Squares with Bound Constraints" on page 4-133
"Example: Jacobian Multiply Function with Linear Least Squares" on page 4-134
"Example: Nonlinear Curve Fitting with lsqcurvefit" on page 4-139

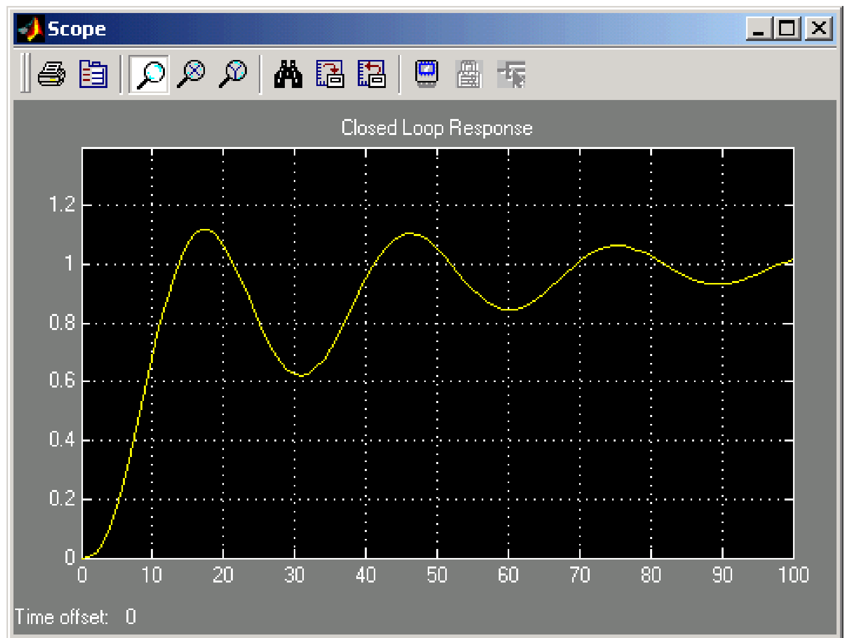
Example: Using lsqnonlin With a Simulink Model

Suppose that you want to optimize the control parameters in the Simulink model `optsim.mdl`. (This model can be found in the `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.



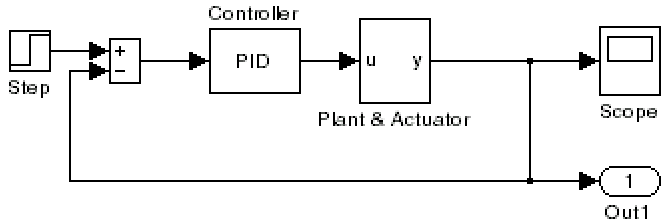
Plant with Actuator Saturation

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 Closed-Loop Response on page 4-127. 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.



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.



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 with Optimization Toolbox software.

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`:

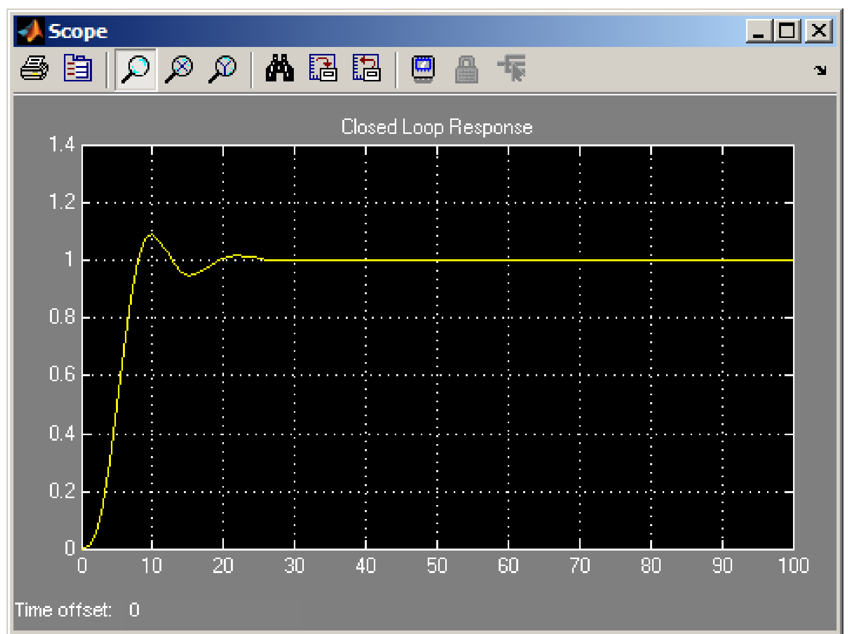
```
function [Kp,Ki,Kd] = runtracklsq
% RUNTRACKLSQ demonstrates using LSQNONLIN with Simulink.

optsim                       % Load the model
pid0 = [0.63 0.0504 1.9688]; % Set initial values
a1 = 3; a2 = 43;             % Initialize model plant variables
options = optimset('Algorithm','levenberg-marquardt',...
   '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

                                         First-Order                   Norm of
 Iteration  Func-count     Residual       optimality      Lambda           step
     0           4          8.66531          0.833         0.01
```

```
     1        8       6.86978          1.2       0.001      1.98067
     2       12        5.6588        0.922      0.0001      3.15662
     3       16       4.57909        0.169      1e-005      6.15155
     4       24        4.4727        0.164         0.1     0.589688
     5       28       4.47066      0.00534        0.01    0.0475163
Optimization terminated: The relative change in the sum-of-squares
of the functions is less than options.TolFun

Kp =
    2.9633

Ki =
    0.1436

Kd =
   13.1386
```

Here is the resulting closed-loop step response.



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 [53].)

Simulink® Response Optimization™ software is recommended for solving multiobjective optimization problems in conjunction with Simulink variable-step solvers. It provides a special numeric gradient computation that works with Simulink and avoids introducing a problem of lack of smoothness.

Example: Nonlinear Least-Squares with Full Jacobian Sparsity Pattern

The trust-region reflective algorithm 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

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

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 `10` 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 trust-region reflective algorithm 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 100s 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.

Example: 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 =
     3
resnorm =
   22.5794
output =
       iterations: 10
        algorithm: 'large-scale: trust-region reflective Newton'
    firstorderopt: 2.7870e-005
     cgiterations: 42
          message: [1x123 char]
```

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 4-24, and `g` is the gradient.

You can improve (decrease) the first-order optimality measure 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 first-order optimality measure decreases:

```
exitflag =
     1
resnorm =
   22.5794
output =
       iterations: 12
        algorithm: 'large-scale: trust-region reflective Newton'
    firstorderopt: 5.5907e-015
     cgiterations: 0
          message: [1x104 char]
```

Example: Jacobian Multiply Function with Linear Least Squares

You can solve a least-squares problem of the form

$$\min_x \|C \cdot x - d\|_2^2$$

such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$, for problems where C is very large, perhaps too large to be stored, by using a Jacobian multiply function.

For example, consider the case where C is a $2n$-by-n matrix based on a circulant matrix. This means the rows of C are shifts of a row vector v. This example has the row vector v with elements of the form $(-1)^{k+1}/k$:

$v = [1, -1/2, 1/3, -1/4, \ldots, -1/n]$,

cyclically shifted:

$$C = \begin{bmatrix} 1 & -1/2 & 1/3 & \dots & -1/n \\ -1/n & 1 & -1/2 & \dots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \dots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \dots & 1 \\ 1 & -1/2 & 1/3 & \dots & -1/n \\ -1/n & 1 & -1/2 & \dots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \dots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \dots & 1 \end{bmatrix}.$$

This least-squares example considers the problem where

$d = [n - 1; n - 2; ...; -n]$,

and the constraints are $-5 \leq x(i) \leq 5$ for $i = 1, ..., n$.

For large enough n, the dense matrix C does not fit into computer memory. ($n = 10{,}000$ is too large on one tested system.)

A Jacobian multiply function has the following syntax:

```
w = jmfcn(Jinfo,Y,flag)
```

`Jinfo` is a matrix the same size as C, used as a preconditioner. If C is too large to fit into memory, `Jinfo` should be sparse. `Y` is a vector or matrix sized so that `C*Y` or `C'*Y` makes sense. `flag` tells `jmfcn` which product to form:

- `flag` $> 0 \Rightarrow$ `w = C*Y`
- `flag` $< 0 \Rightarrow$ `w = C'*Y`
- `flag` $= 0 \Rightarrow$ `w = C'*C*Y`

Since `C` is such a simply structured matrix, it is easy to write a Jacobian multiply function in terms of the vector `v`; i.e., without forming `C`. Each row

of `C*Y` is the product of a shifted version of `v` times `Y`. The following matrix performs one step of the shift: `v` shifts to `v*T`, where

$$T = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{bmatrix}.$$

To compute `C*Y`, compute `v*Y` to find the first row, then shift `v` and compute the second row, and so on.

To compute `C'*Y`, perform the same computation, but use a shifted version of `temp`, the vector formed from the first row of `C'`:

```
temp = [fliplr(v)*T,fliplr(v)*T];
```

To compute `C'*C*Y`, simply compute `C*Y` using shifts of `v`, and then compute `C'` times the result using shifts of `fliplr(v)`.

The `dolsqJac` function in the following code sets up the vector `v` and matrix `T`, and calls the solver `lsqlin`:

```
function [x,resnorm,residual,exitflag,output] = dolsqJac(n)
%
r = 1:n-1; % index for making vectors

T = spalloc(n,n,n); % making a sparse circulant matrix
for m = r
    T(m,m+1)=1;
end
T(n,1) = 1;

v(n) = (-1)^(n+1)/n; % allocating the vector v
v(r) =( -1).^(r+1)./r;

% Now C should be a 2n-by-n circulant matrix based on v,
% but that might be too large to fit into memory.
```

```
r = 1:2*n;
d(r) = n-r;

Jinfo = [speye(n);speye(n)]; % sparse matrix for preconditioning
% This matrix is a required input for the solver;
% preconditioning is not really being used in this example

% Pass the matrix T and vector v so they don't need to be
% computed in the Jacobian multiply function
options = optimset('JacobMult',...
    @(Jinfo,Y,flag)lsqcirculant(Jinfo,Y,flag,T,v));

lb = -5*ones(1,n);
ub = 5*ones(1,n);

[x,resnorm,residual,exitflag,output] = ...
    lsqlin(Jinfo,d,[],[],[],[],lb,ub,[],options);
```

The Jacobian multiply function `lsqcirculant` is as follows:

```
function w = lsqcirculant(Jinfo,Y,flag,T,v)
% This function computes the Jacobian multiply functions
% for a 2n-by-n circulant matrix example

if flag > 0
    w = Jpositive(Y);
elseif flag < 0
    w = Jnegative(Y);
else
    w = Jnegative(Jpositive(Y));
end

    function a = Jpositive(q)
        % Calculate C*q
        temp = v;

        a = zeros(size(q)); % allocating the matrix a
        a = [a;a]; % the result is twice as tall as the input

        for r = 1:size(a,1)
```

```
            a(r,:) = temp*q; % compute the rth row
            temp = temp*T; % shift the circulant
        end
    end

    function a = Jnegative(q)
        % Calculate C'*q
        temp = fliplr(v)*T; % the circulant for C'

        len = size(q,1)/2; % the returned vector is half as long
        % as the input vector
        a = zeros(len,size(q,2)); % allocating the matrix a

        for r = 1:len
            a(r,:) = [temp,temp]*q; % compute the rth row
            temp = temp*T; % shift the circulant
        end
    end
end
```

When `n` = 3000, `C` is an 18,000,000-element dense matrix. Here are the results of the `dolsqJac` function for `n` = 3000 at selected values of `x`, and the `output` structure:

```
[x,resnorm,residual,exitflag,output] = dolsqJac(3000);

Optimization terminated: relative function value changing by
 less than OPTIONS.TolFun.

x(1)
ans =
    5.0000

x(1500)
ans =
   -0.5201

x(3000)
ans =
   -5.0000
```

```
output
output =
       iterations: 16
        algorithm: 'large-scale: trust-region reflective Newton'
   firstorderopt: 5.9351e-005
     cgiterations: 36
          message: [1x87 char]
```

Example: Nonlinear Curve Fitting with lsqcurvefit

`lsqcurvefit` enables you to fit parameterized nonlinear functions to data easily. You can use `lsqnonlin` as well; `lsqcurvefit` is simply a convenient way to call `lsqnonlin` for curve fitting.

In this example, the vector `xdata` represents 100 data points, and the vector `ydata` represents the associated measurements. Generate the data using the following script:

```
rand('twister', 5489);
randn('state', 0);
xdata=-2*log(rand(100,1));
ydata=(ones(100,1) + .1*randn(100,1)) + (3*ones(100,1) + ...
.5*randn(100,1)).*exp((-(2*ones(100,1)+.5*randn(100,1))).*xdata);
```

The modeled relationship between `xdata` and `ydata` is

$$ydata_i = a_1 + a_2 \exp(-a_3 xdata_i) + \varepsilon_i. \quad \textbf{(4-110)}$$

`xdata` is generated by 100 independent samples from an exponential distribution with mean 2. `ydata` is generated from Equation 4-110 using `a` = [1 ;3; 2], perturbed by adding normal deviates with standard deviations [0.1; 0.5; 0.5].

The goal is to find parameters $\hat{a}_i$, i = 1, 2, 3, for the model that best fit the data.

In order to fit the parameters to the data using `lsqcurvefit`, you need to define a fitting function. Define the fitting function `predicted` as an anonymous function:

```
predicted = @(a,xdata) a(1)*ones(100,1)+a(2)*exp(-a(3)*xdata);
```

To fit the model to the data, `lsqcurvefit` needs an initial estimate `a0` of the parameters. Enter

```
a0=[2;2;2];
```

Run the solver `lsqcurvefit` as follows:

```
[ahat,resnorm,residual,exitflag,output,lambda,jacobian]=...
   lsqcurvefit(predicted,a0,xdata,ydata);
```

`lsqcurvefit` calculates the least-squares estimate of $\hat{a}$:

```
ahat =
    1.0259
    2.9777
    2.0077
```

The fitted values `ahat` are within 3% of `a` = [1; 3; 2].

If you have Statistics Toolbox™ software, use the `nlparci` function to generate confidence intervals for the `ahat` estimate.

Multiobjective Optimization

In this section...

"Definition" on page 4-141

"Algorithms" on page 4-142

Definition

There are two Optimization Toolbox multiobjective solvers: `fgoalattain` and `fminimax`.

- `fgoalattain` addresses the problem of reducing a set of nonlinear functions $F_i(x)$ below a set of goals F^*_i. Since there are several functions $F_i(x)$, it is not always clear what it means to solve this problem, especially when you cannot achieve all the goals simultaneously. Therefore, the problem is reformulated to one that is always well-defined.

  The *unscaled goal attainment problem* is to minimize the maximum of $F_i(x) - F^*_i$.

  There is a useful generalization of the unscaled problem. Given a set of positive weights w_i, the *goal attainment problem* tries to find x to minimize the maximum of

$$\frac{F_i(x) - F_i^*}{w_i}. \tag{4-111}$$

  This minimization is supposed to be accomplished while satisfying all types of constraints: $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

  If you set all weights equal to 1 (or any other positive constant), the goal attainment problem is the same as the unscaled goal attainment problem. If the F^*_i are positive, and you set all weights as $w_i = F^*_i$, the goal attainment problem becomes minimizing the relative difference between the functions $F_i(x)$ and the goals F^*_i.

  In other words, the goal attainment problem is to minimize a slack variable γ, defined as the maximum over i of the expressions in Equation 4-111. This implies the expression that is the formal statement of the goal attainment problem:

$$\min_{x,\gamma} \gamma$$

such that $F(x) - w\gamma \le F^*$, $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

- `fminimax` addresses the problem of minimizing the maximum of a set of nonlinear functions, subject to all types of constraints:

  $$\min_{x} \max_{i} F_i(x)$$

  such that $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

  Clearly, this problem is a special case of the unscaled goal attainment problem, with $F^*_i = 0$ and $w_i = 1$.

Algorithms

Goal Attainment Method

This section describes the goal attainment method of Gembicki [16]. This

method uses a set of design goals, $F^* = \left\{F_1^*, F_2^*, \ldots, F_m^*\right\}$, 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 the 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 formulation

$$\underset{\gamma \in \Re,\ x \in \Omega}{\text{minimize}}\ \gamma \qquad \textbf{(4-112)}$$

such that $F_i(x) - w_i\gamma \le F_i^*, \quad i = 1,\ldots,m.$

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 overachievement 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 ([10] and [11]).

The goal attainment method is represented geometrically in the figure below in two dimensions.

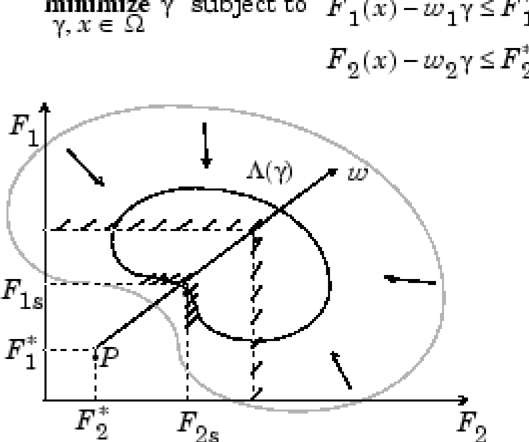


Figure 4-6: Geometrical Representation of the Goal Attainment Method

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 the 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 [36]). In goal attainment programming there might be a more appropriate merit function, which you can achieve by posing Equation 4-112 as the minimax problem

$$\underset{x\in\Re^n}{\text{minimize}}\ \max_i \{\Lambda_i\}, \tag{4-113}$$

where $\Lambda_i = \dfrac{F_i(x) - F_i^*}{w_i},\ \ i = 1,...,m.$

Following the argument of Brayton et al. [2] for minimax optimization using SQP, using the merit function of Equation 4-47 for the goal attainment problem of Equation 4-113 gives

$$\psi(x,\gamma) = \gamma + \sum_{i=1}^{m} r_i \cdot \max\left\{0, F_i(x) - w_i\gamma - F_i^*\right\}. \tag{4-114}$$

When the merit function of Equation 4-114 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.,

$$\psi(x) = \max_i \Lambda_i. \tag{4-115}$$

A problem in the goal attainment method is that it is common to use a weighting coefficient equal to 0 to incorporate hard constraints. The merit function of Equation 4-115 then becomes infinite for arbitrary violations of the constraints.

To overcome this problem while still retaining the features of Equation 4-115, the merit function is combined with that of Equation 4-48, giving the following:

$$\psi(x) = \sum_{i=1}^{m} \begin{cases} r_i \cdot \max\left\{0, F_i(x) - w_i\gamma - F_i^*\right\} & \text{if } w_i = 0 \\ \max_i \Lambda_i,\ i = 1,...,m & \text{otherwise.} \end{cases} \tag{4-116}$$

Another feature that can be exploited in SQP is the objective function γ. From the KKT equations 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 4-31.

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 Equation 4-47.

Minimizing the Maximum Objective

`fminimax` uses a goal attainment method. It takes goals of 0, and weights of 1. With this formulation, the goal attainment problem becomes

$$\min_i \max_x \left(\frac{f_i(x) - goal_i}{weight_i} \right) = \min_i \max_x f_i(x),$$

which is the minimax problem.

Parenthetically, you might expect `fminimax` to turn the multiobjective function into a single objective. The function

$$f(x) = \max(F_1(x),...F_j(x))$$

is a single objective function to minimize. However, it is not differentiable, and Optimization Toolbox objectives are required to be smooth. Therefore the minimax problem is formulated as a smooth goal attainment problem.

Multiobjective Optimization Examples

In this section...

"Example: Using fminimax with a Simulink Model" on page 4-147
"Example: Signal Processing Using fgoalattain" on page 4-150

Example: Using fminimax with a Simulink Model

Another approach to optimizing the control parameters in the Simulink model shown in Plant with Actuator Saturation on page 4-126 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` $\leq$ `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

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.
        % Objective TRACKMMOBJ is called before this
        % constraint function, so yout is current.
        c = -yout(20:100)+.95;
        ceq=[];
    end
end
```

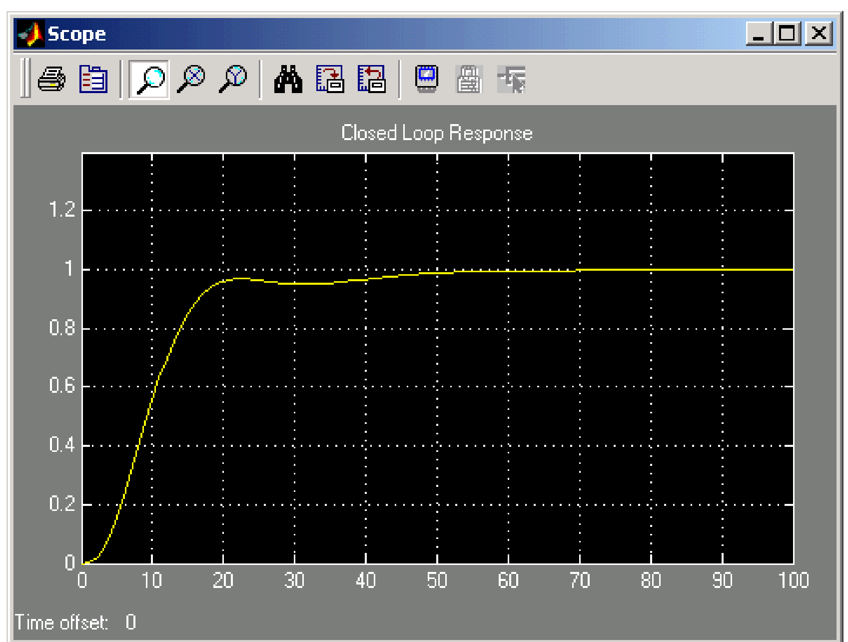
When you run the code, it returns the following results:

```
[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: magnitude of directional derivative in search
 direction less than 2*options.TolFun and maximum constraint violation
  is less than options.TolCon.
  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 figure Closed-Loop Response Using fminimax on page 4-150.

This solution differs from the solution obtained in “Example: Using lsqnonlin With a Simulink Model” on page 4-126 because you are solving different problem formulations.



Closed-Loop Response Using fminimax

Example: Signal Processing Using fgoalattain

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

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

$$\begin{aligned} H(f) &= \sum_{n=0}^{2M} h(n)e^{-j2\pi fn} \\ &= A(f)e^{-j2\pi fM}, \\ A(f) &= \sum_{n=0}^{M-1} a(n)\cos(2\pi fn), \end{aligned} \tag{4-117}$$

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 4-142). 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

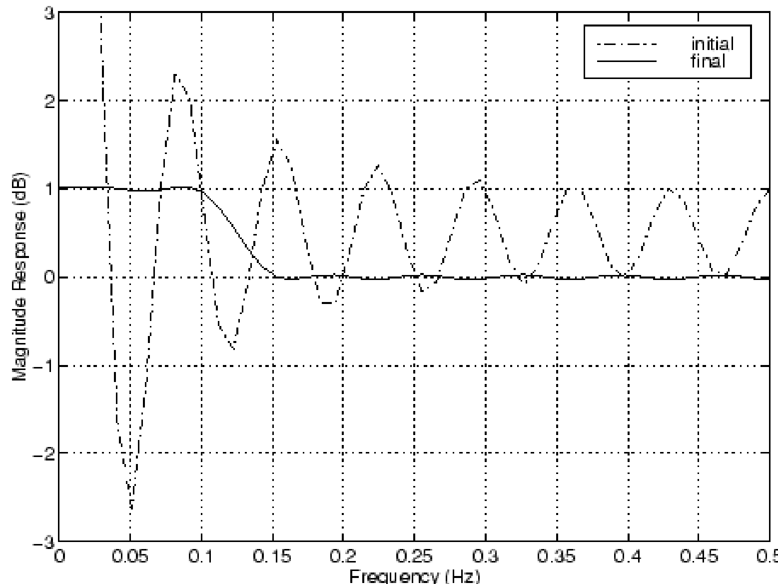
```
function y = filtmin(a,w)
n = length(a);
y = cos(w'*(0:n-1)*2*pi)*a ;
```

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 (Magnitude Response with Initial and Final Magnitude Coefficients on page 4-153). Note that you could use the `firpm` function in Signal Processing Toolbox™ software to design this filter.



Magnitude Response with Initial and Final Magnitude Coefficients

Equation Solving

In this section...

"Definition" on page 4-154
"Trust-Region Dogleg Method" on page 4-155
"Trust-Region Reflective fsolve Algorithm" on page 4-157
"Levenberg-Marquardt Method" on page 4-160
"Gauss-Newton Method" on page 4-160
"\ Algorithm" on page 4-161
"fzero Algorithm" on page 4-161

Definition

Given a set of n nonlinear functions $F_i(x)$, where n is the number of components of the vector x, the goal of equation solving is to find a vector x that makes all $F_i(x) = 0$.

The `fsolve` function has four algorithms for solving systems of nonlinear multidimensional equations:

- Trust-region dogleg
- Trust-region-reflective
- Levenberg-Marquardt
- Gauss-Newton

All but the Gauss-Newton method are large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-45.

The `fzero` function solves a single one-dimensional equation.

The `\` function solves systems of linear equations.

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 4-3) 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

$$\min_d m(d) = \frac{1}{2}\left\| M(x_k + d) \right\|_2^2 = \frac{1}{2}\left\| F(x_k) + J(x_k)d \right\|_2^2$$
$$= \frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T J(x_k)^T J(x_k) d. \quad \textbf{(4-118)}$$

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

$$\min_d \left[\frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T J(x_k)^T J(x_k) d \right], \quad \textbf{(4-119)}$$

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 [4], and Nocedal [31].

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 Equation 4-119. For a detailed description, see Powell [34].

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 Equation 4-118.

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.

Trust-Region Reflective fsolve Algorithm

Many of the methods used in Optimization Toolbox solvers 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,

$$\min_{s}\{q(s),\ s \in N\}. \tag{4-120}$$

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

$$\min\left\{\frac{1}{2}s^T Hs + s^T g \text{ such that } \|Ds\| \le \Delta\right\}, \tag{4-121}$$

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 $\|\ .\ \|$ is the 2-norm. Good algorithms exist for solving Equation 4-121 (see [48]); 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 Equation 4-121. 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 Equation 4-121, have been proposed in the literature ([42] and [50]). The approximation approach followed in Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). Once the subspace S has been computed, the work to solve Equation 4-121 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 solver defines S as the linear space spanned by s_1 and s_2, 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

$$H \cdot s_2 = -g, \tag{4-122}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{4-123}$$

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 sketch of unconstrained minimization using trust-region ideas is now easy to give:

1 Formulate the two-dimensional trust-region subproblem.

2 Solve Equation 4-121 to determine the trial step s.

3 If $f(x + s) < f(x)$, then $x = x + s$.

4 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 [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat 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 Gradient Method

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.

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., $d^T Hd \leq 0$. 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 4-3.

Levenberg-Marquardt Method

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left(J(x_k)^T J(x_k) + \lambda_k I\right) d_k = -J(x_k)^T F(x_k), \tag{4-124}$$

or, optionally, of the equations

$$\left(J(x_k)^T J(x_k) + \lambda_k diag\left(J(x_k)^T J(x_k)\right)\right) d_k = -J(x_k)^T F(x_k), \tag{4-125}$$

where the scalar λ_k controls both the magnitude and direction of d_k. Set option `ScaleProblem` to `'none'` to choose Equation 4-124, and set `ScaleProblem` to `'Jacobian'` to choose Equation 4-125.

When λ_k is zero, the direction d_k is identical to that of the Gauss-Newton method. As λ_k tends to infinity, d_k tends towards the steepest descent direction, with magnitude tending to zero. 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 direction.

Gauss-Newton Method

One approach to solving this problem is to use a Nonlinear Least-Squares solver, such as those described in "Least Squares (Model Fitting)" on page 4-116. Since the assumption is the system has a root, it would have a small residual; therefore, using the Gauss-Newton Method is effective. In this case, each iteration solves a linear least-squares problem, as described in Equation 4-109, to find the search direction. (See "Gauss-Newton Method" on page 4-122 for more information.)

Gauss-Newton Implementation

The Gauss-Newton algorithm is the same as that for least-squares optimization. It is described in "Gauss-Newton Method" on page 4-122.

\ Algorithm

This algorithm is described in the MATLAB arithmetic operators section for `\` (`mldivide`).

fzero Algorithm

`fzero` attempts to find the root of a scalar function *f* of a scalar variable *x*.

`fzero` looks for an interval around an initial point such that *f*(*x*) changes sign. If you give an initial interval instead of an initial point, `fzero` checks to make sure *f*(*x*) has different signs at the endpoints of the interval. The initial interval must be finite; it cannot contain ±`Inf`.

`fzero` uses a combination of interval bisection, linear interpolation, and inverse quadratic interpolation in order to locate a root of *f*(*x*). See `fzero` for more information.

Equation Solving Examples

In this section...

"Example: Nonlinear Equations with Analytic Jacobian" on page 4-162
"Example: Nonlinear Equations with Finite-Difference Jacobian" on page 4-165
"Example: Nonlinear Equations with Jacobian" on page 4-166
"Example: Nonlinear Equations with Jacobian Sparsity Pattern" on page 4-169

Example: Nonlinear Equations with Analytic Jacobian

This example demonstrates the use of the default large-scale `fsolve` algorithm (see "Large-Scale vs. Medium-Scale Algorithms" on page 2-45). 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 of $F(x) = 0$, 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

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

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

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

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:

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

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. Set `Display` to `'iter'` to see the solver's progress. 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
```

Example: 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.

```
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.

Example: 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, $F(x) = 0$, you can use the trust-region reflective algorithm available in `fsolve`, a large-scale algorithm ("Large-Scale vs. Medium-Scale Algorithms" on page 2-45).

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;
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',...
    'Algorithm','trust-region-reflective',...
    'Jacobian','on','PrecondBandWidth',0);
[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 trust-region-dogleg, so it is necessary to specify `'Algorithm'` as `'trust-region-reflective'` in the `options` argument in order to select the trust-region-reflective algorithm. 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
Iteration  Func-count     f(x)            step          optimality   CG-iterations
    0          1          1011                                19
```

```
    1          2         16.1942         7.91898          2.35            3
    2          3       0.0228027         1.33142         0.291            3
    3          4     0.000103359       0.0433329        0.0201            4
    4          5     7.3792e-007       0.0022606      0.000946            4
    5          6    4.02299e-010     0.000268381      4.12e-005           5
Optimization terminated: 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. Setting `PrecondBandWidth` to 0 in `options` means 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).

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',...
   'Algorithm','trust-region-reflective','PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case the output is

```
                                         Norm of      First-order
Iteration  Func-count     f(x)           step          optimality   CG-iterations
    0          1            1011                            19
    1          2         16.0839         7.92496          1.92            1
    2          3       0.0458181          1.3279         0.579            1
    3          4     0.000101184       0.0631898        0.0203            2
    4          5    3.16615e-007      0.00273698       0.00079            2
    5          6    9.72481e-010      0.00018111      5.82e-005           2
Optimization terminated: 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 Gradient Method” on page 4-23.

Setting `PrecondBandWidth` to `Inf` (this is the default) means that the solver uses Cholesky factorization rather than PCG.

Example: 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,...
    'Algorithm','trust-region-reflective','PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case, the output displayed is

```
                                         Norm of      First-order
Iteration  Func-count     f(x)           step          optimality   CG-iterations
    0          5             1011                            19
    1         10          16.0839        7.92496           1.92             1
    2         15        0.0458179         1.3279          0.579             1
    3         20      0.000101184      0.0631896         0.0203             2
    4         25      3.16616e-007    0.00273698        0.00079             2
    5         30      9.72483e-010    0.00018111       5.82e-005            2
Optimization terminated: 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. For example, 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,...
 'Algorithm','trust-region-reflective','PrecondBandWidth',inf);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

and the resulting display is

```
                                          Norm of      First-order
Iteration  Func-count     f(x)            step          optimality   CG-iterations
    0          5             1011                            19
    1         10          15.9018          7.92421            1.89            0
    2         15        0.0128161          1.32542          0.0746            0
    3         20      1.73502e-008       0.0397923        0.000196            0
    4         25      1.10732e-018    4.55495e-005        2.74e-009           0
Optimization terminated: first-order optimality less than OPTIONS.TolFun,
 and no negative/zero curvature detected in trust region model.
```

When the sparse direct solvers are used, the CG iteration is 0 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.

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] 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.

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

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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.

[40] Byrd, R.H., J. C. Gilbert, and J. Nocedal, “A Trust Region Method Based on Interior Point Techniques for Nonlinear Programming,” *Mathematical Programming*, Vol 89, No. 1, pp. 149–185, 2000.

[41] Byrd, R.H., Mary E. Hribar, and Jorge Nocedal, “An Interior Point Algorithm for Large-Scale Nonlinear Programming, SIAM Journal on Optimization,” *SIAM Journal on Optimization*, Vol 9, No. 4, pp. 877–900, 1999.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Coleman, T.F. and A. Verma, "A Preconditioned Conjugate Gradient Approach to Linear Equality Constrained Minimization," submitted to *Computational Optimization and Applications*.

[47] Mehrotra, S., "On the Implementation of a Primal-Dual Interior Point Method," *SIAM Journal on Optimization*, Vol. 2, pp 575-601, 1992.

[48] 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.

[49] 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.

[50] Steihaug, T., "The Conjugate Gradient Method and Trust Regions in Large Scale Optimization," *SIAM Journal on Numerical Analysis*, Vol. 20, pp 626-637, 1983.

[51] Waltz, R. A. , J. L. Morales, J. Nocedal, and D. Orban, "An interior algorithm for nonlinear optimization that combines line search and trust region steps," *Mathematical Programming*, Vol 107, No. 3, pp. 391–408, 2006.

[52] 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.

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

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

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

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

[57] 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.

5

Parallel Computing for Optimization

- "Parallel Computing in Optimization Toolbox Functions" on page 5-2
- "Using Parallel Computing with fmincon, fgoalattain, and fminimax" on page 5-5
- "Improving Performance with Parallel Computing" on page 5-8

Parallel Computing in Optimization Toolbox Functions

In this section...

"Parallel Optimization Functionality" on page 5-2
"Parallel Estimation of Gradients" on page 5-2
"Nested Parallel Functions" on page 5-4

Parallel Optimization Functionality

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

The Optimization Toolbox solvers `fmincon`, `fgoalattain`, and `fminimax` can automatically distribute the numerical estimation of gradients of objective functions and nonlinear constraint functions to multiple processors. These solvers use parallel gradient estimation under the following conditions:

- You have a license for Parallel Computing Toolbox™ software.
- The option `GradObj` is set to `'off'`, or, if there is a nonlinear constraint function, the option `GradConstr` is set to `'off'`. Since `'off'` is the default value of these options, you don't have to set them with `optimset`; just don't set them both to `'on'`.
- Parallel computing is enabled with `matlabpool`, a Parallel Computing Toolbox function.
- The option `UseParallel` is set to `'always'`. The default value of this option is `'never'`.

When these conditions hold, the solvers compute estimated gradients in parallel.

Parallel Estimation of Gradients

One subroutine was made parallel in the functions `fmincon`, `fgoalattain`, and `fminimax`: the subroutine that estimates the gradient of the objective function and constraint functions. This calculation involves computing

function values at points near the current location x. Essentially, the calculation is

$$\nabla f(x) \approx \left[\frac{f(x+\Delta_1 e_1) - f(x)}{\Delta_1}, \frac{f(x+\Delta_2 e_2) - f(x)}{\Delta_2}, \ldots, \frac{f(x+\Delta_n e_n) - f(x)}{\Delta_n} \right],$$

where

- f represents objective or constraint functions
- e_i are the unit direction vectors
- Δ_i is the size of a step in the e_i direction

To estimate $\nabla f(x)$ in parallel, Optimization Toolbox solvers distribute the evaluation of $(f(x + \Delta_i e_i) - f(x))/\Delta_i$ to extra processors.

`fmincon` uses the parallel subroutine only with the active-set algorithm. `fgoalattain` and `fminimax` always use the parallel subroutine.

Parallel Central Differences

You can choose to have gradients estimated by central finite differences instead of the default forward finite differences. The basic central finite difference formula is

$$\nabla f(x) \approx \left[\frac{f(x+\Delta_1 e_1) - f(x-\Delta_1 e_1)}{2\Delta_1}, \ldots, \frac{f(x+\Delta_n e_n) - f(x-\Delta_n e_n)}{2\Delta_n} \right].$$

This takes twice as many function evaluations as forward finite differences, but is usually much more accurate. Central finite differences work in parallel exactly the same as forward finite differences.

Enable central finite differences by using `optimset` to set the `FinDiffType` option to `'central'`. To use forward finite differences, set the `FinDiffType` option to `'forward'`.

Nested Parallel Functions

Solvers employ the Parallel Computing Toolbox function `parfor` to perform parallel estimation of gradients. `parfor` does not work in parallel when called from within another `parfor` loop. Therefore, you cannot simultaneously use parallel gradient estimation and parallel functionality within your objective or constraint functions.

Suppose, for example, your objective function `userfcn` calls `parfor`, and you wish to call `fmincon` in a loop. Suppose also that the conditions for parallel gradient evaluation of `fmincon`, as given in "Parallel Optimization Functionality" on page 5-2, are satisfied. Figure When parfor Runs In Parallel on page 5-4 shows three cases:

1. The outermost loop is `parfor`. Only that loop runs in parallel.
2. The outermost `parfor` loop is in `fmincon`. Only `fmincon` runs in parallel.
3. The outermost `parfor` loop is in `userfcn`. `userfcn` can use `parfor` in parallel.

Bold indicates the function that runs in parallel

①
```
...
parfor (i=1:10)
  x(i)=fmincon(@userfcn,...)
  ...
end
```
Only the outermost parfor loop runs in parallel

②
```
...
for (i=1:10)
  x(i)=fmincon(@userfcn,...)
  ...
end
```
If UseParallel = 'always' fmincon runs in parallel

③
```
...
for (i=1:10)
  x(i)=fmincon(@userfcn,...)
  ...
end
```
If UseParallel = 'never' userfcn can use parfor in parallel

When parfor Runs In Parallel

Using Parallel Computing with fmincon, fgoalattain, and fminimax

In this section...

"Using Parallel Computing with Multicore Processors" on page 5-5
"Using Parallel Computing with a Multiprocessor Network" on page 5-6
"Testing Parallel Computations" on page 5-7

Using Parallel Computing with Multicore Processors

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

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

- Enter

  ```
  matlabpool open 2
  ```

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

- Enter

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

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

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

```
matlabpool close
```

Using Parallel Computing with a Multiprocessor Network

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

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

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

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

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

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

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

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

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

   ```
   filename not found.
   ```

3. At the command line enter

   ```
   matlabpool open conf
   ```

   or

   ```
   matlabpool open conf n
   ```

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

4. Enter

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

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

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

```
matlabpool close
```

Testing Parallel Computations

To test see if a problem runs correctly in parallel,

1. Try your problem without parallel computation to ensure that it runs properly serially. Make sure this is successful (gives correct results) before going to the next test.

2. Set `UseParallel` to `'always'`, and ensure `matlabpool` is closed. Your problem runs `parfor` serially, with loop iterations in reverse order from a `for` loop. Make sure this is successful (gives correct results) before going to the next test.

3. Set `UseParallel` to `'always'`, and open `matlabpool`. Unless you have a multicore processor or a network set up, you won't see any speedup. This testing is simply to verify the correctness of the computations.

Remember to call your solver using an options structure to test or use parallel functionality.

Improving Performance with Parallel Computing

In this section...

"Factors That Affect Speed" on page 5-8
"Factors That Affect Results" on page 5-8
"Searching for Global Optima" on page 5-9

Factors That Affect Speed

Some factors may affect the speed of execution of parallel processing:

- `parfor` overhead. There is overhead in calling `parfor` instead of `for`. If function evaluations are fast, this overhead could become appreciable.
- No nested `parfor` loops. This is described in "Nested Parallel Functions" on page 5-4. `parfor` does not work in parallel when called from within another `parfor` loop. If you have programmed your objective or constraint functions to take advantage of parallel processing, the limitation of no nested `parfor` loops may cause a solver to run more slowly than you expect. In particular, the parallel computation of finite differences takes precedence, since that is an outer loop. This causes any parallel code within the objective or constraint functions to execute serially.
- Passing parameters. Parameters are automatically passed to worker machines during the execution of parallel computations. If there are a large number of parameters, or they take a large amount of memory, passing them may slow the execution of your computation.
- Contention for resources: network and computing. If the network of worker machines has low bandwidth or high latency, computation could be slowed.

Factors That Affect Results

Some factors may affect numerical results when using parallel processing. There are more caveats related to `parfor` listed in the "Limitations" section of the Parallel Computing Toolbox documentation.

- Persistent or global variables. If your objective or constraint functions use persistent or global variables, these variables may take different values

on different worker processors. Furthermore, they may not be cleared properly on the worker processors.

- Accessing external files. External files may be accessed in an unpredictable fashion during a parallel computation. The order of computations is not guaranteed during parallel processing, so external files may be accessed in unpredictable order, leading to unpredictable results.
- Accessing external files. If two or more processors try to read an external file simultaneously, the file may become locked, leading to a read error, and halting the execution of the optimization.
- If your objective function calls Simulink, results may be unreliable with parallel gradient estimation.
- Noncomputational functions, such as `input`, `plot`, and `keyboard`, might behave badly when used in objective or constraint functions. When called in a `parfor` loop, these functions are executed on worker machines. This can cause a worker to become nonresponsive, since it is waiting for input.
- `parfor` does not allow `break` or `return` statements.

Searching for Global Optima

To search for global optima, one approach is to evaluate a solver from a variety of initial points. If you distribute those evaluations over a number of processors using the `parfor` function, you disable parallel gradient estimation, since `parfor` loops cannot be nested. Your optimization usually runs more quickly if you distribute the evaluations over all the processors, rather than running them serially with parallel gradient estimation, so disabling parallel estimation probably won't slow your computation. If you have more processors than initial points, though, it is not clear whether it is better to distribute initial points or to enable parallel gradient estimation.

6

External Interface

ktrlink: An Interface to KNITRO Libraries

In this section...

"What Is ktrlink?" on page 6-2
"Installation and Configuration" on page 6-2
"Example Using ktrlink" on page 6-4
"Setting Options" on page 6-7
"Sparse Matrix Considerations" on page 6-9

What Is ktrlink?

`ktrlink` calls Ziena Optimization's KNITRO® libraries in order to perform an optimization. `ktrlink` can address constrained and unconstrained problems. To use `ktrlink`, you must purchase a copy of KNITRO libraries from Ziena Optimization, Inc. (`http://www.ziena.com/`).

Use `ktrlink` the same as any other Optimization Toolbox function.

`ktrlink`'s syntax is similar to `fmincon`'s. The main differences are:

- `ktrlink` has additional options input for KNITRO libraries so you can access its options.
- `ktrlink` has no provision for obtaining a returned Hessian or gradient, since KNITRO software doesn't return them.
- Sparse matrix representations differ between KNITRO software and MATLAB.

Furthermore, many returned flags and messages differ from `fmincon`'s, because they are returned directly from KNITRO libraries.

Installation and Configuration

The system requirements for MATLAB and KNITRO software may differ. Check the system requirements for both products before attempting to use `ktrlink`. For recent and planned MATLAB platform support, see

`http://www.mathworks.com/support/sysreq/roadmap.html`

Optimization Toolbox software versions 4.0 and 4.1 work with version 5.2 of KNITRO libraries. Contact Ziena Optimization, Inc. (`http://www.ziena.com/`) if you have questions regarding other versions of the KNITRO libraries.

Note `ktrlink` is not compatible with the Solaris™ 64-bit architecture.

Perform the following steps to configure your system to use `ktrlink`:

1. Install MATLAB and the KNITRO libraries on your system.
2. Set the system path to include the KNITRO libraries (see "Setting the System Path to Include KNITRO Libraries" on page 6-3). Make sure to perform this step before starting MATLAB.
3. Start MATLAB.

Setting the System Path to Include KNITRO Libraries

In order to use `ktrlink`, you need to tell MATLAB where the KNITRO binary file (`libknitro.so`, `libknitro.dylib`, `knitro520.dll`, or a similar file) resides. You do this by setting a system-wide environment variable. Enter the following system-level commands. Replace *<file_absolute_path>* with the full path to your KNITRO libraries:

- Linux:

  ```
  setenv LD_LIBRARY_PATH <file_absolute_path>:$LD_LIBRARY_PATH
  ```

- Macintosh:

  1. In **Terminal** (available in the **Utilities** folder of the **Applications** folder) enter the following:

     ```
     edit /etc/profile
     ```

  2. Add the following line to the end of the file:

```
export DYLD_LIBRARY_PATH=<file_absolute_path>:$DYLD_LIBRARY_PATH
```

- Windows:
  1. At the Windows desktop, right-click **My Computer** (Windows XP) or **Computer** (Vista).
  2. Select **Properties**.
  3. Click the **Advanced** tab (Windows XP) or **Advanced System Settings** (Vista).
  4. Click **Environment Variables**.
  5. Under **System variables**, edit the **Path** variable to add the KNITRO library directory.

Check if the installation was successful by starting MATLAB and running the following command:

```
[x fval] = ktrlink(@(x)cos(x),1)
```

If you receive an error message, check your system path, and make sure the KNITRO libraries are on the path. When installed correctly, `ktrlink` returns `x = 3.1416`, `fval = -1`.

Example Using ktrlink

1. This example uses the same constraint function as the example in "Nonlinear Constraints" on page 2-14. The constraint function is the intersection of the interior of an ellipse with the region above a parabola:

```
function [c ceq gradc gradceq]=ellipseparabola(x)
% Inside the ellipse bounded by (-3<x<3),(-2<y<2)
% Above the line y=x^2-1
c(1) = x(1)^2/9 + x(2)^2/4 - 1;% ellipse
c(2) = x(1)^2 - x(2) - 1;% parabola
ceq = [];

if nargout > 2
    gradc = [2*x(1)/9, 2*x(1);...
             x(2)/2, -1];
    gradceq = [];
```

```
end
```

2 The objective function is a tilted `sinh`:

```
function [f gradf]=sinhtilt(x)

A=[2,1;1,2];
m=[1,1];
f=sinh(x'*A*x/100) + sinh(m*A*x/10);

if nargout > 1
    gradf=cosh(x'*A*x/100)*(A*x)/50;
    gradf=gradf+cosh(m*A*x/10)*[3;3]/10;
end
```

3 Set the options so that `ktrlink` has iterative display and uses the gradients included in the objective and constraint functions:

```
ktropts = optimset('Display','iter',...
          'GradConstr','on','GradObj','on');
```

4 Run the optimization starting at [0;0], using the structure `ktropts`:

```
[x fval flag] = ktrlink(@sinhtilt,[0;0],...
           [],[],[],[],[],[],@ellipseparabola,ktropts)
```

KNITRO software returns the following output:

```
======================================
           KNITRO 5.2
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
======================================


algorithm:    1
hessopt:      2
honorbnds:    1
outlev:       4
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt from AUTO to 2.
KNITRO changing honorbnds to 0 (because there are no bounds).
```

```
Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                    2
    bounded below:                      0
    bounded above:                      0
    bounded below and above:            0
    fixed:                              0
    free:                               2
Number of constraints:                  2
    linear equalities:                  0
    nonlinear equalities:               0
    linear inequalities:                0
    nonlinear inequalities:             2
    range:                              0
Number of nonzeros in Jacobian:         4
Number of nonzeros in Hessian:          3

 Iter(maj/min)  Res   Objective      Feas err    Opt err     ||Step||    CG its
-------------- ---  ------------  ----------  ----------  ----------  -------
      0/      0 ---  0.000000e+000  0.000e+000
      1/      1 Acc -1.371022e-001  0.000e+000  1.976e-001  3.432e-001        0
      2/      2 Acc -3.219187e-001  0.000e+000  1.003e-001  4.748e-001        0
      3/      3 Acc -3.442829e-001  0.000e+000  6.287e-002  8.198e-002        0
      4/      4 Acc -3.598395e-001  8.358e-003  2.989e-002  1.904e-001        0
      5/      5 Acc -3.619202e-001  6.453e-003  2.812e-003  1.481e-001        0
      6/      6 Acc -3.591357e-001  0.000e+000  9.999e-004  1.901e-002        0
      7/      7 Acc -3.599184e-001  0.000e+000  2.441e-004  3.406e-003        0
      8/      8 Acc -3.599195e-001  0.000e+000  2.000e-004  1.749e-004        0
      9/      9 Acc -3.601176e-001  0.000e+000  1.076e-005  6.823e-004        0
     10/     10 Acc -3.601176e-001  0.000e+000  2.000e-006  9.351e-007        0
     11/     11 Acc -3.601196e-001  0.000e+000  2.001e-008  7.893e-006        0

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.

Final Statistics
----------------
Final objective value               = -3.60119556291455e-001
Final feasibility error (abs / rel) =   0.00e+000 / 0.00e+000
```

```
Final optimality error  (abs / rel) =   2.00e-008 / 2.00e-008
# of iterations (major / minor)     =         11 /       11
# of function evaluations           =         13
# of gradient evaluations           =         12
Total program time (secs)           =       0.081 (     0.156 CPU time)
Time spent in evaluations (secs)    =       0.079

=========================================================================

x =
   -0.5083
   -0.7416

fval =
   -0.3601

flag =
           0
```

Note Exit flags have different meanings for `ktrlink` and `fmincon`. Flag 0 for KNITRO libraries means the first-order optimality condition was satisfied; for `fmincon`, the corresponding flag is 1. For more information about the output, see the KNITRO documentation at `http://www.ziena.com/`.

Setting Options

`ktrlink` takes up to two options inputs: one in `fmincon` format, and another in KNITRO format. You can use either or both types of options. If you use both types of options, MATLAB reads only four `fmincon`-format options: `HessFcn`, `HessMult`, `HessPattern`, and `JacobPattern`. KNITRO options override `fmincon`-format options.

To use KNITRO options, create an options text file, whose format can be found in the KNITRO documentation. For example, if you have a KNITRO options file named `knitropts`, and an `fmincon`-format options structure named `ktropts`, you can pass them both by calling `ktrlink` like this:

```
[x fval] = ktrlink(@bigtoleft,[-1,-1,-1],...
           [],[],[],[],[],[],@twocone,ktropts,'knitropts')
```

If `knitropts` resides in a different directory, pass the path to the file. For example:

```
[x fval] = ktrlink(@bigtoleft,[-1,-1,-1],...
           [],[],[],[],[],[],@twocone,ktropts,...
           'C:\Documents\Knitro\knitropts')
```

The following shows how `fmincon`-format options are mapped to KNITRO options.

fmincon Option	KNITRO Option	Default Value
Algorithm	algorithm	'interior-point'
AlwaysHonorConstraints	honorbounds	'bounds'
Display	outlev	'none'
FinDiffType	gradopt	'forward'
GradConstr	gradopt	'off'
GradObj	gradopt	'off'
HessFcn	hessopt	[ ]
MaxIter	maxit	10000
TolFun	opttol	1.00E–06
TolX	xtol	1.00E–15
Hessian, LBFGS pairs	hessopt, lmsize	'bfgs'
HessMult	hessopt	[ ]
HessPattern		[ ]
InitBarrierParam	bar_initmu	0.1
InitTrustRegionRadius	delta	sqrt(numberOfVariables)
JacobPattern		[ ]
MaxProjCGIter	maxcgit	2*(numberOfVariables–numberOfEqualities)
ObjectiveLimit	objrange	–1.0E20
ScaleProblem	scale	'obj-and-constr'
SubProblemAlgorithm	algorithm	'ldl-factorization'
TolCon	feastol	1.00E-06

Note KNITRO libraries allow you to pass simultaneously either the gradients of the objective function and all nonlinear constraints, or no gradients. Therefore, setting `GradObj = 'on'` and `GradConstr = 'off'` is inconsistent. If you attempt to pass inconsistent options, `ktrlink` warns you, and treats all gradients as if they had not been passed.

Sparse Matrix Considerations

When the Hessian of the Lagrangian is sparse, or the Jacobian of the nonlinear constraints is sparse, `ktrlink` makes use of the sparsity structure to speed the optimization and use less memory doing so.

`ktrlink` handles sparse matrices differently than other MATLAB functions. If you choose to use sparse matrices, `ktrlink` requires a sparsity pattern for nonlinear constraint Jacobians and for Hessians. The next two sections give the details of the sparsity patterns for nonlinear constraint Jacobians and for Hessians.

Sparsity Pattern for Nonlinear Constraints

The sparsity pattern for constraint Jacobians is a matrix. You pass the matrix as the `JacobPattern` option. The structure of the matrix follows.

Let c denote the vector of m nonlinear inequality constraints, and let ceq denote the vector of m_2 nonlinear equality constraints. If there are n dimensions, the Jacobian is an $(m + m_2)$-by-n matrix. The Jacobian pattern J is

$$J_{i,j} = \begin{cases} 1 \text{ if } \dfrac{\partial c_i(x)}{\partial x_j} \neq 0 \\ 0 \text{ if } \dfrac{\partial c_i(x)}{\partial x_j} \equiv 0 \end{cases}, \quad 1 \le i \le m,$$

$$J_{i+m,j} = \begin{cases} 1 \text{ if } \dfrac{\partial ceq_i(x)}{\partial x_j} \neq 0 \\ 0 \text{ if } \dfrac{\partial ceq_i(x)}{\partial x_j} \equiv 0 \end{cases}, \quad 1 \le i \le m_2.$$

In other words, the ith row of the Jacobian pattern corresponds to the gradient of c_i. Inequality gradients lie above equality gradients (they have lower row numbers).

All that matters for the Jacobian pattern is whether or not the entries are zero. You can pass single-precision numbers, doubles, or logical true or false. You can pass the sparsity pattern as a MATLAB sparse matrix. If you have a large sparse matrix of constraints, it is more efficient to pass the pattern as a sparse matrix. Linear constraint matrices are automatically passed as sparse matrices.

The gradient of the constraints, calculated in the constraint function, has the transpose of the Jacobian pattern. For more information about the form of constraint gradients, see "Nonlinear Constraints" on page 2-14.

Sparsity Pattern for Hessians

The Hessian is the matrix of second derivatives of the Lagrangian:

$$H_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} + \sum_k \lambda_k \frac{\partial^2 c_k}{\partial x_i \partial x_j} + \sum_m \lambda_m \frac{\partial^2 ceq_m}{\partial x_i \partial x_j}.$$

Give the matrix H as a full or sparse matrix of zero and nonzero elements. The elements can be single-precision numbers, doubles, or logical true or false.

The Hessian is a symmetric matrix. You can pass just the upper triangular pattern, or pass the whole matrix pattern.

7

Argument and Options Reference

- "Function Arguments" on page 7-2
- "Optimization Options" on page 7-7

Function Arguments

In this section...

"Input Arguments" on page 7-2
"Output Arguments" on page 7-5

Input Arguments

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.	bintprog, 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.	bintprog, 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.	bintprog, 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

Input Arguments (Continued)

Argument	Description	Used by Functions
goal	Vector of values that the objectives attempt to attain. The vector is the same length as the number of objectives.	fgoalattain
H	The matrix of coefficients for the quadratic terms in the quadratic equation x'*H*x+f'*x. H must be symmetric.	quadprog
lb, ub	Lower and upper bound vectors (or matrices). The arguments are normally the same size as x. However, if lb has fewer elements than x, say only m, then only the first m elements in x are bounded below; upper bounds in ub can be defined in the same manner. 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. "Passing Extra Parameters" on page 2-17 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

Input Arguments (Continued)

Argument	Description	Used by Functions
`options`	An structure that defines options used by the optimization functions. For information about the options, see "Optimization Options" on page 7-7 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. "Passing Extra Parameters" on page 2-17 explains how to parameterize `seminfcon`, if necessary. See the function reference pages for `fseminf` for more information on `seminfcon`.	`fseminf`
`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 a finite 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

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. See the function reference pages 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.	bintprog, fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, linprog, quadprog
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

Output Arguments (Continued)

Argument	Description	Used by Functions
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

In this section...

"Options Structure" on page 7-7

"Output Function" on page 7-17

"Plot Functions" on page 7-26

Options Structure

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
- I — Interior-point method only

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 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.

Optimization Options

Option Name	Description	L, M, B, I	Used by Functions
Algorithm	Chooses the algorithm used by the solver.	B, I	fmincon, fsolve, lsqcurvefit, lsqnonlin
AlwaysHonorConstraints	The default 'bounds' ensures that bound constraints are satisfied at every iteration. Turn off by setting to 'none'.	I	fmincon
BranchStrategy	Strategy bintprog uses to select branch variable.	M	bintprog
DerivativeCheck	Compare user-supplied analytic derivatives (gradients or Jacobian, depending on the selected solver) 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
DiffMaxChange	Maximum change in variables for finite differencing.	B, I	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin
DiffMinChange	Minimum change in variables for finite differencing.	B, I	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
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.
FinDiffType	Finite differences, used to estimate gradients, are either 'forward' (the default) , or 'central' (centered), which takes twice as many function evaluations but should be more accurate. 'central' differences might violate bounds during their evaluation in fmincon interior-point evaluations if the AlwaysHonorConstraints option is set to 'none'.	M, I	fgoalattain, fmincon, fminimax, fminunc, fseminf
FunValCheck	Check whether objective function and constraints values are valid. 'on' displays an error when the objective function or constraints return a value that is complex, NaN, or Inf. **Note** FunValCheck does not return an error for Inf when used with fminbnd, fminsearch, or fzero, which handle Inf appropriately. 'off' displays no error.	B	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, lsqnonlin

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
`GoalsExactAchieve`	Specify the number of objectives required for the objective `fun` to equal the goal `goal`. Objectives should be partitioned into the first few elements of `F`.	M	`fgoalattain`
`GradConstr`	User-defined gradients for the nonlinear constraints.	M, I	`fgoalattain`, `fmincon`, `fminimax`
`GradObj`	User-defined gradients for the objective functions.	B	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`
`HessFcn`	Function handle to a user-supplied Hessian (see "Hessian" on page 9-41).	I	`fmincon`
`Hessian`	If `'user-supplied'`, 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, I	`fmincon`, `fminunc`
`HessMult`	Handle to a user-supplied Hessian multiply function. Ignored unless `Hessian` is `'user-supplied'` or `'on'`.	L, I	`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`
`InitBarrierParam`	Initial barrier value.	I	`fmincon`

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
`InitialHessMatrix`	Initial quasi-Newton matrix.	M	`fminunc`
`InitialHessType`	Initial quasi-Newton matrix type.	M	`fminunc`
`InitTrustRegionRadius`	Initial radius of the trust region.	I	`fmincon`
`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`	User-defined Jacobian multiply function. Ignored unless `Jacobian` is `'on'` for `fsolve`, `lsqcurvefit`, and `lsqnonlin`.	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	`fminunc`, `fsolve`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`LevenbergMarquardt`	Choose Gauss-Newton algorithm by setting `LevenbergMarquardt` to `'off'` and `LargeScale` to `'off'`.	M	`fsolve`, `lsqcurvefit`, `lsqnonlin`

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
`LineSearchType`	Line search algorithm choice.	M	`fsolve`, `lsqcurvefit`, `lsqnonlin`
`MaxFunEvals`	Maximum number of function evaluations allowed.	B	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`
`MaxIter`	Maximum number of iterations allowed.	B	All but `fzero` and `lsqnonneg`
`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`
`MaxProjCGIter`	A tolerance for the number of projected conjugate gradient iterations; this is an inner iteration, not the number of iterations of the algorithm.	I	`fmincon`
`MaxRLPIter`	Maximum number of iterations of linear programming relaxation method allowed.	M	`bintprog`

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
MaxSQPIter	Maximum number of iterations of sequential quadratic programming method allowed.	M	fgoalattain, fmincon, fminimax
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
NodeSearchStrategy	Search strategy that bintprog uses.	M	bintprog
NonlEqnAlgorithm	Specify the Gauss-Newton algorithm for solving nonlinear equations by setting NonlEqnAlgorithm to 'gn' and LargeScale to 'off'.	M	fsolve
ObjectiveLimit	If the objective function value goes below ObjectiveLimit and the iterate is feasible, then the iterations halt.	I	fmincon
OutputFcn	Specify one or more user-defined functions that the optimization function calls at each iteration. See "Output Function" on page 7-17.	B	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, lsqnonlin

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying @optimplotx plots the current point; @optimplotfunccount plots the function count; @optimplotfval plots the function value; @optimplotconstrviolation plots the maximum constraint violation; @optimplotresnorm plots the norm of the residuals; @optimplotstepsize plots the step size; @optimplotfirstorderopt plots the first-order of optimality. See "Plot Functions" on page 7-26.	B	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, and lsqnonlin. See the individual function reference pages for the values that apply.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG. Setting to 'Inf' uses a direct factorization instead of CG.	L	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog
RelLineSrchBnd	Relative bound on line search step length.	M	fgoalattain, fmincon, fminimax, fseminf
RelLineSrchBndDuration	Number of iterations for which the bound specified in RelLineSrchBnd should be active.	M	fgoalattain, fmincon, fminimax, fseminf

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
ScaleProblem	For fmincon, the default 'obj-and-constr' causes the algorithm to normalize all constraints and the objective function. Disable by setting to 'none'. For the other solvers, when using the Algorithm option 'levenberg-marquardt', setting the ScaleProblem option to 'jacobian' sometimes helps the solver on badly-scaled problems.	L, I	fmincon, fsolve, lsqcurvefit, lsqnonlin
Simplex	If 'on', function uses the simplex algorithm.	M	linprog
SubproblemAlgorithm	Determines how the iteration step is calculated.	I	fmincon
TolCon	Termination tolerance on the constraint violation.	B	bintprog, fgoalattain, fmincon, fminimax, fseminf
TolConSQP	Constraint violation tolerance for the inner SQP iteration.	M	fgoalattain, fmincon, fminimax, fseminf
TolFun	Termination tolerance on the function value.	B	bintprog, fgoalattain, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, linprog (L only), lsqcurvefit, lsqlin (L only), lsqnonlin, quadprog (L only)

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
`TolPCG`	Termination tolerance on the PCG iteration.	L	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`TolProjCG`	A relative tolerance for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.	I	`fmincon`
`TolProjCGAbs`	Absolute tolerance for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.	I	`fmincon`
`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`

Optimization Options (Continued)

Option Name	Description	L, M, B, I	Used by Functions
`TypicalX`	Array that specifies typical magnitude of array of parameters `x`. The size of the array is equal to the size of `x0`, the starting point.	B	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`
`UseParallel`	When `'always'`, applicable solvers estimate gradients in parallel. Disable by setting to `'never'`.	M	`fgoalattain`, `fmincon`, `fminimax`.

Output Function

The `Outputfcn` field of the `options` structure specifies one or more functions that an optimization function calls at each iteration. Typically, you might use an output function to plot points at each iteration or to display optimization quantities from the algorithm. Using an output function you can view, but not set, optimization quantities. 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);
   ```

   specifies `OutputFcn` to be the handle to `outfun`. To specify more than one output function, use the syntax

   ```
   options = optimset('OutputFcn',{@outfun, @outfun2});
   ```

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

See “Output Functions” on page 2-33 for an example of an output function.

"Passing Extra Parameters" on page 2-17 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 7-18 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 7-25 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 7-25 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 2-47. 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, L, M, and B indicate:

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

optimValues Fields

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
cgiterations	Number of conjugate gradient iterations at current optimization iteration.	fmincon (L), fsolve (L), lsqcurvefit (L), lsqnonlin (L)	CG-iterations See “Displaying Iterative Output” on page 2-47.
constrviolation	Maximum constraint violation.	fgoalattain (M), fmincon (M), fminimax (M), fseminf (M)	max constraint See “Displaying Iterative Output” on page 2-47.

optimValues Fields (Continued)

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 7-24.	`fmincon` (L), `fsolve` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	None
`directionalderivative`	Directional derivative in the search direction.	`fgoalattain` (M), `fmincon` (M), `fminimax` (M), `fminunc` (M), `fseminf` (M), `fsolve` (M), `lsqcurvefit` (M), `lsqnonlin` (M)	`Directional derivative` See "Displaying Iterative Output" on page 2-47.
`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), `fsolve` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	`First-order optimality` See "Displaying Iterative Output" on page 2-47.

optimValues Fields (Continued)

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), `fsolve` (B), `fzero` (B), `fseminf` (M), `lsqcurvefit` (B), `lsqnonlin` (B)	`F-count` See "Displaying Iterative Output" on page 2-47.
`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), `fsolve` (B), `fzero` (B)	`f(x)` See "Displaying Iterative Output" on page 2-47.
`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), `fsolve` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	None
`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), `fsolve` (B), `fseminf` (M), `fzero` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	`Iteration` See "Displaying Iterative Output" on page 2-47.

optimValues Fields (Continued)

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`lambda`	The Levenberg-Marquardt parameter, `lambda`, at the current iteration. See "Levenberg-Marquardt Method" on page 4-121.	`fsolve` (B, Levenberg-Marquardt and Gauss-Newton algorithms), `lsqcurvefit` (B, Levenberg-Marquardt and Gauss-Newton algorithms), `lsqnonlin` (B, Levenberg-Marquardt and Gauss-Newton algorithms)	`Lambda`
`positivedefinite`	`0` if algorithm detects negative curvature while computing Newton step. `1` otherwise.	`fmincon` (L), `fsolve` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	None
`procedure`	Procedure messages.	`fgoalattain` (M), `fminbnd` (B), `fmincon` (M), `fminimax` (M), `fminsearch` (B), `fseminf` (M), `fzero` (B)	`Procedure` See "Displaying Iterative Output" on page 2-47.
`ratio`	Ratio of change in the objective function to change in the quadratic approximation.	`fmincon` (L), `fsolve` (L), `lsqcurvefit` (L), `lsqnonlin` (L)	None
`residual`	The residual vector. For `fsolve`, `residual` means the 2-norm of the residual squared.	`lsqcurvefit` (B), `lsqnonlin` (B), `fsolve` (B)	`Residual` See "Displaying Iterative Output" on page 2-47.

optimValues Fields (Continued)

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`resnorm`	2-norm of the residual squared.	`lsqcurvefit` (B), `lsqnonlin` (B)	`Resnorm` See "Displaying Iterative Output" on page 2-47.
`searchdirection`	Search direction.	`fgoalattain` (M), `fmincon` (M), `fminimax` (M), `fminunc` (M), `fseminf` (M), `fsolve` (M), `lsqcurvefit` (M), `lsqnonlin` (M)	None
`stepaccept`	Status of the current trust-region step. Returns true if the current trust-region step was successful, and false if the trust-region step was unsuccessful.	`fsolve` (L, NonlEqn-Algorithm='dogleg')	None
`stepsize`	Current step size (displacement in x). Final value equals optimization function output `output.stepsize`.	`fgoalattain` (M), `fmincon` (B), `fminimax` (M), `fminunc` (B), `fseminf` (M), `fsolve` (B), `lsqcurvefit` (B), `lsqnonlin` (B)	`Step-size` or `Norm of Step` See "Displaying Iterative Output" on page 2-47.
`trustregionradius`	Radius of trust region.	`fmincon` (L), `fsolve` (L, M), `lsqcurvefit`, `lsqnonlin` (L)	`Trust-region radius` See "Displaying Iterative Output" on page 2-47.

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 clicks 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');
```

Plot Functions

The `PlotFcns` field of the `options` structure specifies one or more functions that an optimization function calls at each iteration to plot various measures of progress while the algorithm executes. The structure of a plot function is the same as that for an output function. For more information on writing and calling a plot function, see "Output Function" on page 7-17.

To view and modify a predefined plot function listed for `PlotFcns` in the previous table, you can open it in the MATLAB Editor. For example, to view the M-file corresponding to the norm of residuals, type:

```
edit optimplotresnorm.m
```

8

Function Reference

Minimization (p. 8-2)	Solve minimization problems
Equation Solving (p. 8-2)	Equation solving
Least Squares (Curve Fitting) (p. 8-3)	Solve least-squares problems
GUI (p. 8-3)	Open Optimization Tool to select solver, optimization options, and run problems
Utilities (p. 8-4)	Get and set optimization options

Minimization

`bintprog`	Solve binary integer programming problems
`fgoalattain`	Solve multiobjective goal attainment problems
`fminbnd`	Find minimum of single-variable function on fixed interval
`fmincon`	Find minimum of constrained nonlinear multivariable function
`fminimax`	Solve minimax constraint problem
`fminsearch`	Find minimum of unconstrained multivariable function using derivative-free method
`fminunc`	Find minimum of unconstrained multivariable function
`fseminf`	Find minimum of semi-infinitely constrained multivariable nonlinear function
`ktrlink`	Find minimum of constrained or unconstrained nonlinear multivariable function using KNITRO third-party libraries
`linprog`	Solve linear programming problems
`quadprog`	Solve quadratic programming problems

Equation Solving

See \ for solving linear equations of the form $Ax = b$.

`fsolve`	Solve system of nonlinear equations
`fzero`	Find root of continuous function of one variable

Least Squares (Curve Fitting)

See \ for minimizing $\|Ax - b\|$.

`lsqcurvefit`	Solve nonlinear curve-fitting (data-fitting) problems in least-squares sense
`lsqlin`	Solve constrained linear least-squares problems
`lsqnonlin`	Solve nonlinear least-squares (nonlinear data-fitting) problems
`lsqnonneg`	Solve nonnegative least-squares constraint problem

GUI

`optimtool`	Tool to select solver, optimization options, and run problems

Utilities

`color`	Column partition for sparse finite differences
`fzmult`	Multiplication with fundamental nullspace basis
`gangstr`	Zero out "small" entries subject to structural rank
`optimget`	Optimization options values
`optimset`	Create or edit optimization options structure

9

Functions — Alphabetical List

bintprog

Purpose

Solve binary integer programming problems

Equation

Solves binary integer programming problems of the form

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ x \text{ binary.} \end{cases}$$

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 = bintprog(problem)
[x,fval] = bintprog(...)
[x,fval,exitflag] = bintprog(...)
[x,fval,exitflag,output] = bintprog(...)
```

Description

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

$$\min_x f^T x.$$

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

$$\min_x f^T x \text{ such that } A \cdot x \le b.$$

`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 = bintprog(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-3.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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 9-4.

`[x,fval,exitflag,output] = bintprog(...)` returns a structure `output` that contains information about the optimization. See "Output Arguments" on page 9-4.

Input Arguments

The following table lists the input arguments for `bintprog`. "Function Arguments" on page 7-2 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.	
`problem`	`f`	Linear objective function vector `f`
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`x0`	Initial point for `x`
	`solver`	`'bintprog'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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	Optimization algorithm used
	branchStrategy	Strategy used to select branch variable—see "Options" on page 9-5
	nodeSearchStrategy	Strategy used to select next node in search tree—see "Options" on page 9-5
	message	Exit message

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 7-7 for detailed information.

BranchStrategy	Strategy the algorithm uses to select the branch variable in the search tree — see "Branching" on page 4-108. 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).
`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.
`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
`NodeDisplayInterval`	Node display interval
`NodeSearchStrategy`	Strategy the algorithm uses to select the next node to search in the search tree — see “Branching” on page 4-108. 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.
`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

For more information, see "bintprog Algorithm" on page 4-108

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.

x =
      1
      1
      0
      0
```

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.

See Also

`linprog`, `optimset`, `optimtool`

For more details about the `bintprog` algorithm, see "Binary Integer Programming" on page 4-108. For another example of integer programming, see "Binary Integer Programming Example" on page 4-111.

Purpose

Column partition for sparse finite differences

Syntax

```
group = color(J,P)
```

Description

`group = color(J,P)` returns a partition of the column corresponding to a coloring of the column-intersection graph. `GROUP(I) = J` means column `I` is colored `J`.

All columns belonging to a color can be estimated in a single finite difference.

fgoalattain

Purpose

Solve multiobjective goal attainment problems

Equation

Finds the minimum of a problem specified by

$$\underset{x,\gamma}{\text{minimize}}\,\gamma \quad \text{such that} \begin{cases} F(x) - weight \cdot \gamma \le goal \\ c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub. \end{cases}$$

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 = fgoalattain(problem)
[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.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective functions and nonlinear constraint functions, if necessary.

`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 = fgoalattain(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-12.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

[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.

Note If the specified input bounds for a problem are inconsistent, the output x is x0 and the output fval is [].

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into fgoalattain. This section provides function-specific details for fun, goal, nonlcon, options, weight, and problem:

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);
```

If the user-defined values for x and F are matrices, they are converted to a vector using linear indexing.

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. 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)).

Note Setting `GradObj` to `'on'` is effective only when there is no nonlinear constraint, or when the nonlinear constraint has `GradConstr` set to `'on'` as well. This is because internally the objective is folded into the constraints, so the solver needs both gradients (objective and constraint) supplied in order to avoid estimating a gradient.

`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)`. “Nonlinear Constraints” on page 2-14 explains how to “conditionalize” the gradients for use in solvers that do not accept supplied gradients.

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)`).

Note Setting `GradConstr` to `'on'` is effective only when `GradObj` is set to `'on'` as well. This is because internally the objective is folded into the constraint, so the solver needs both gradients (objective and constraint) supplied in order to avoid estimating a gradient.

Note Because Optimization Toolbox functions only accept inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

“Passing Extra Parameters” on page 2-17 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

`options`	"Options" on page 9-19 provides the function-specific details for the `options` values.
`weight`	A weighting vector to control the relative underattainment 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 a component of the `weight` vector to zero will cause the corresponding goal constraint to be treated as a hard constraint rather than as a goal constraint. An alternative method to set a hard constraint is to use the input argument `nonlcon`.

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`).

problem	objective	Vector of objective functions
	x0	Initial point for x
	goal	Goals to attain
	weight	Relative importance factors of goals
	Aineq	Matrix for linear inequality constraints
	bineq	Vector for linear inequality constraints
	Aeq	Matrix for linear equality constraints
	beq	Vector for linear equality constraints
	lb	Vector of lower bounds
	ub	Vector of upper bounds
	nonlcon	Nonlinear constraint function
	solver	'fgoalattain'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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. attainfactor contains the value of γ at the solution. A negative value of γ indicates overattainment in the goals.
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

Field	Description
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
lssteplength	Size of line search step relative to search direction
stepsize	Final displacement in x
algorithm	Optimization algorithm used
firstorderopt	Measure of first-order optimality
constrviolation	Maximum of nonlinear constraint functions
message	Exit message

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 7-7 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; 'notify' displays output only if the function does not converge;'final' (default) displays just the final output.

`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations but should be more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. So, for example, it may take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
`FunValCheck`	Check whether objective function and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is `complex`, `Inf`, or `NaN`. `'off'` displays no error.
`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 user-defined objective function. See the preceding description of `fun` to see how to define the gradient in `fun`.
`MaxFunEvals`	Maximum number of function evaluations allowed.

`MaxIter`	Maximum number of iterations allowed.
`MaxSQPIter`	Maximum number of SQP iterations allowed.
`MeritFunction`	Use goal attainment/minimax merit function if set to `'multiobj'`. Use `fmincon` merit function if set to `'singleobj'`.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx` plots the current point; `@optimplotfunccount` plots the function count; `@optimplotfval` plots the function value; `@optimplotconstrviolation` plots the maximum constraint violation; `@optimplotstepsize` plots the step size; `@optimplotfirstorderopt` plots the first-order of optimality.
`RelLineSrchBnd`	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in `x` satisfies $\lvert\Delta x(i)\rvert \le \text{relLineSrchBnd} \cdot \max(\lvert x(i)\rvert, \lvert typicalx(i)\rvert)$. This option provides control over the magnitude of the displacements in `x` for cases in which the solver takes steps that are considered too large.

RelLineSrchBndDuration	Number of iterations for which the bound specified in RelLineSrchBnd should be active (default is 1).
TolCon	Termination tolerance on the constraint violation.
TolConSQP	Termination tolerance on inner iteration SQP constraint violation.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.
TypicalX	Typical x values. The length of the vector is equal to the number of elements in x0, the starting point.
UseParallel	When 'always', estimate gradients in parallel. Disable by setting to 'never'.

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.

$$A = \begin{bmatrix} -0.5 & 0 & 0 \\ 0 & -2 & 10 \\ 0 & 1 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ -2 & 2 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

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),...
    K0,goal,weight,[],[],[],[],lb,ub,[],options)
```

You can run this example by using the demonstration script `goaldemo`. (From the MATLAB Help browser or the MathWorks Web site documentation, you can click the demo name to display the demo.) 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 in "Multiobjective Optimization" on page 4-141.

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 in "Sequential Quadratic Programming (SQP)" on page 4-27. 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] and [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 4-142 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`.

See also "SQP Implementation" on page 4-29 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.

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.

See Also

`@ (function_handle)`, `fmincon`, `fminimax`, `optimset`, `optimtool`

For more details about the `fgoalattain` algorithm, see "Multiobjective Optimization" on page 4-141. For another example of goal attainment, see "Multiobjective Optimization Examples" on page 4-147.

fminbnd

Purpose

Find minimum of single-variable function on fixed interval

Equation

Finds a minimum for a problem specified by

$$\min_{x} f(x) \text{ such that } x_1 < x < x_2.$$

x, x_1, and x_2 are scalars and $f(x)$ is a function that returns a scalar.

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
x = fminbnd(problem)
[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.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective function, if necessary.

`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`. `fun` is a function handle for either an M-file function or an anonymous function.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fminbnd(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-29.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fminbnd`. This section provides function-specific details for `fun`, `options`, and `problem`:

`fun`

The function to be minimized. `fun` is a function handle for 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^2),x1,x2);
```

`options`

"Options" on page 9-30 provides the function-specific details for the `options` values.

problem	f	Objective function
	x1	Left endpoint
	x2	Right endpoint
	solver	'fminbnd'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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	Optimization 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 7-7 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 an error when the objective function returns a value that is complex or NaN. 'off' displays no error.
MaxFunEvals	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations allowed.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying @optimplotx plots the current point; @optimplotfunccount plots the function count; @optimplotfval plots the function value.
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. Unless the left endpoint x_1 is very close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$. If the minimum actually occurs at x_1 or x_2, `fminbnd` returns an interior point at a distance of no more than `2*TolX` from x_1 or x_2, where `TolX` is the termination tolerance. See [1] or [2] for details about the algorithm.

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.

References

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

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

See Also

`@ (function_handle)`, `fminsearch`, `fmincon`, `fminunc`, `optimset`, `optimtool`, “Anonymous Functions”

fmincon

Purpose

Find minimum of constrained nonlinear multivariable function

Equation

Finds the minimum of a problem specified by

$$\min_{x} f(x) \text{ such that } \begin{cases} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub, \end{cases}$$

x, b, beq, lb, and *ub* are vectors, *A* and *Aeq* are matrices, $c(x)$ and $ceq(x)$ are functions that return vectors, and $f(x)$ is a function that returns a scalar. $f(x)$, $c(x)$, and $ceq(x)$ can be nonlinear functions.

Syntax

```
x = fmincon(fun,x0,A,b)
x = fmincon(fun,x0,A,b,Aeq,beq)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fmincon(problem)
[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*.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`x = fmincon(fun,x0,A,b)` starts at `x0` and attempts to find a minimizer `x` of 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` and `A*x ≤ b`. If no inequalities exist, set `A = []` and `b = []`.

`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`. If no equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` is unbounded below, set `lb(i) = -Inf`, and if `x(i)` is unbounded above, set `ub(i) = Inf`.

`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`. If no bounds exist, set `lb = []` and/or `ub = []`.

`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. If there are no nonlinear inequality or equality constraints, set `nonlcon = []`.

`x = fmincon(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-36.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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 9-41.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Components of `x0` that violate the bounds `lb ≤ x ≤ ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

Input Arguments

"Function Arguments" on page 7-2 describes the arguments passed to `fmincon`. "Options" on page 9-135 provides the function-specific details for the `options` values. This section provides function-specific details for `fun`, `nonlcon`, and `problem`.

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`. `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 `fun` must return the gradient vector `g(x)` in the second output argument.

If the Hessian matrix can also be computed *and* the `Hessian` option is `'on'` via `options = optimset('Hessian','user-supplied')` *and* the `Algorithm` option is `trust-region-reflective`, `fun` must return the Hessian value `H(x)`, a symmetric matrix, in a third output argument. See “Writing Objective Functions” on page 2-4 for details.

If the Hessian matrix can be computed and the `Algorithm` option is `interior-point`, there are several ways to pass the Hessian to `fmincon`. For more information, see “Hessian” on page 9-41.

nonlcon — The function that computes the nonlinear inequality constraints `c(x)≤ 0` and the nonlinear equality constraints `ceq(x) = 0`. `nonlcon` accepts a vector `x` and returns the two vectors `c` and `ceq`. `c` is a vector that contains the nonlinear inequalities evaluated at `x`, and `ceq` is a vector that contains the nonlinear equalities evaluated at `x`. `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 `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`. For more information, see “Nonlinear Constraints” on page 2-14.

Note Because Optimization Toolbox functions only accept inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

“Passing Extra Parameters” on page 2-17 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

problem	objective	Objective function
	x0	Initial point for x
	Aineq	Matrix for linear inequality constraints
	bineq	Vector for linear inequality constraints
	Aeq	Matrix for linear equality constraints
	beq	Vector for linear equality constraints
	lb	Vector of lower bounds
	ub	Vector of upper bounds
	nonlcon	Nonlinear constraint function
	solver	'fmincon'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 describes 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 measure was less than options.TolFun, and maximum constraint violation was less than options.TolCon.
2	Change in x was less than options.TolX.
3	Change in the objective function value was less than options.TolFun.

	4	Magnitude of the search direction was less than 2*`options.TolX` and constraint violation was less than `options.TolCon`.
	5	Magnitude of directional derivative in search direction was less than 2*`options.TolFun` and maximum constraint violation was less than `options.TolCon`.
	0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
	-1	The output function terminated the algorithm.
	-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
`lssteplength`	Size of line search step relative to search direction (active-set algorithm only)
`constrviolation`	Maximum of constraint violations (active-set and interior-point algorithms)
`stepsize`	Length of last displacement in `x` (active-set and interior-point algorithms)
`algorithm`	Optimization algorithm used
`cgiterations`	Total number of PCG iterations (trust-region-reflective and interior-point algorithms)
`firstorderopt`	Measure of first-order optimality
`message`	Exit message

Hessian

`fmincon` uses a Hessian, the second derivatives of the Lagrangian (see Equation 2-2), namely,

$$\nabla_{xx}^2 L(x,\lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x). \qquad \textbf{(9-1)}$$

There are three algorithms used by `fmincon`, and each one handles Hessians differently:

- The `active-set` algorithm does not accept a user-supplied Hessian. It computes a quasi-Newton approximation to the Hessian of the Lagrangian.
- The `trust-region-reflective` can accept a user-supplied Hessian as the final output of the objective function. Since this algorithm has only bounds or linear constraints, the Hessian of the Lagrangian is

same as the Hessian of the objective function. See "Writing Objective Functions" on page 2-4 for details on how to pass the Hessian to `fmincon`. Indicate that you are supplying a Hessian by

```
options = optimset('Hessian','user-supplied');
```

If you do not pass a Hessian, the algorithm computes a finite-difference approximation.

- The `interior-point` algorithm can accept a user-supplied Hessian as a separately defined function—it is not computed in the objective function. The syntax is

  ```
  hessian = hessianfcn(x, lambda)
  ```

  `hessian` is an n-by-n matrix, sparse or dense, where n is the number of variables. `lambda` is a structure with the Lagrange multiplier vectors associated with the nonlinear constraints:

  ```
  lambda.ineqnonlin
  lambda.eqnonlin
  ```

  `fmincon` computes the structure `lambda`. `hessianfcn` must calculate the sums in Equation 9-1. Indicate that you are supplying a Hessian by

  ```
  options = optimset('Hessian','user-supplied',...
                     'HessFcn',@hessianfcn);
  ```

The `interior-point` algorithm has several more options for Hessians:

- `options = optimset('Hessian','bfgs');`

  `fmincon` calculates the Hessian by a dense quasi-Newton approximation.

- `options = optimset('Hessian',{'lbfgs',positive integer});`

  `fmincon` calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation. The positive integer specifies how many past iterations should be remembered.

- `options = optimset('Hessian','lbfgs');`

  `fmincon` calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation. The default memory, 10 iterations, is used.

- ```
  options = optimset('Hessian','fin-diff-grads',...
  'SubproblemAlgorithm','cg','GradObj','on',...
  'GradConstr','on');
  ```

  `fmincon` calculates a Hessian-times-vector product by finite differences of the gradient(s). You must supply the gradient of the objective function, and also gradients of nonlinear constraints if they exist.

- ```
  options = optimset('Hessian','on',...
  'SubproblemAlgorithm','cg', 'HessMult',@HessMultFcn]);
  ```

  `fmincon` uses a Hessian-times-vector product. You must supply the function `HessMultFcn`, which returns an n-by-1 vector. The `HessMult` option enables you to pass the result of multiplying the Hessian by a vector without calculating the Hessian.

The `'HessMult'` option for the interior-point algorithm has a different syntax than that of the trust-region-reflective algorithm. The syntax for the interior-point algorithm is

```
W = HessMultFcn(x,lambda,v);
```

The result `W` should be the product `H*v`, where `H` is the Hessian at `x`, `lambda` is the Lagrange multiplier (computed by `fmincon`), and `v` is a vector. In contrast, the syntax for the trust-region-reflective algorithm does not involve `lambda`:

```
W = HessMultFcn(H,v);
```

Again, the result `W = H*v`. `H` is the function returned in the third output of the objective function (see "Writing Objective Functions" on page 2-4), and `v` is a vector. `H` does not have to be the Hessian; rather, it can be any function that enables you to calculate `W = H*v`.

Options

Optimization options used by fmincon. Some options apply to all algorithms, and others are relevant for particular algorithms. You can use optimset to set or change the values of these fields in the structure options. See "Optimization Options" on page 7-7 for detailed information.

fmincon uses one of three algorithms: active-set, interior-point, or trust-region-reflective. You choose the algorithm at the command line with optimset. For example:

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

The default trust-region-reflective (formerly called large-scale) requires:

- A gradient to be supplied in the objective function
- 'GradObj' to be set to 'on'
- Either bound constraints or linear equality constraints, but not both

If these conditions are not all satisfied, the 'active-set' algorithm (formerly called medium-scale) is the default.

The 'active-set' algorithm is not a large-scale algorithm.

All Algorithms

These options are used by all three algorithms:

Algorithm	Choose the optimization algorithm.
DerivativeCheck	Compare user-supplied derivatives (gradients of the objective and constraints) to finite-difference approximates.
Diagnostics	Display diagnostic information about the function to be minimized.
DiffMaxChange	Maximum change in variables for finite differencing.
DiffMinChange	Minimum change in variables for finite differencing.

`Display`	Level of display. • `'off'` displays no output • `'iter'` displays output at each iteration • `'notify'` displays output only if the function does not converge • `'final'` (default) displays just the final output
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. `'off'` displays no error.
`GradObj`	Gradient for the objective function defined by the user. You must provide the gradient to use the trust-region-reflective method. This option is not required for the active-set and interior-point methods. See the preceding description of `fun` to see how to define the gradient in `fun`.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.

`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. • `@optimplotx` plots the current point • `@optimplotfunccount` plots the function count • `@optimplotfval` plots the function value • `@optimplotconstrviolation` plots the maximum constraint violation • `@optimplotstepsize` plots the step size • `@optimplotfirstorderopt` plots the first-order of optimality
`TolFun`	Termination tolerance on the function value.
`TolCon`	Termination tolerance on the constraint violation.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Trust-Region-Reflective Algorithm

These options are used by the trust-region-reflective 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)`
	where `Hinfo` contains a matrix used to compute `H*Y`.
	The first argument must be the same as the third argument returned by the objective function `fun`, for example:
	`[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. See "Passing Extra Parameters" on page 2-17 for information on how to supply values for any additional parameters that `hmfun` needs.
	Note `'Hessian'` must be set to `'on'` for `Hinfo` to be passed from `fun` to `hmfun`.

	See “Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints” on page 4-64 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 trust-region-reflective method in `fmincon` can approximate `H` via sparse finite differences (of the gradient) provided that you supply the *sparsity structure* of `H`—i.e., locations of the nonzeros—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. For more information, see “Preconditioned Conjugate Gradient Method” on page 4-23.
`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. Setting `PrecondBandWidth` to `'Inf'` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.
`TolPCG`	Termination tolerance on the PCG iteration.

Active-Set Algorithm

These options are used only by the active-set algorithm:

`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations but should be more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. So, for example, it may take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
`MaxSQPIter`	Maximum number of SQP iterations allowed.
`RelLineSrchBnd`	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in x satisfies $\|\Delta x(i)\| \leq \text{relLineSrchBnd} \cdot \max(\|x(i)\|,\|typicalx(i)\|)$. This option provides control over the magnitude of the displacements in x for cases in which the solver takes steps that are considered too large.
`RelLineSrchBndDuration`	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is 1).

TolConSQP	Termination tolerance on inner iteration SQP constraint violation.
UseParallel	When 'always', estimate gradients in parallel. Disable by setting to 'never'.

Interior-Point Algorithm

These options are used by the interior-point algorithm:

AlwaysHonorConstraints	The default 'bounds' ensures that bound constraints are satisfied at every iteration. Disable by setting to 'none'.
FinDiffType	Finite differences, used to estimate gradients, are either 'forward' (the default), or 'central' (centered). 'central' takes twice as many function evaluations but should be more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. So, for example, it may take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds. However, 'central' differences might violate bounds during their evaluation if the AlwaysHonorConstraints option is set to 'none'.
HessFcn	Function handle to a user-supplied Hessian (see "Hessian" on page 9-41).
Hessian	Chooses how fmincon calculates the Hessian (see "Hessian" on page 9-41).

`HessMult`	Handle to a user-supplied function that gives a Hessian-times-vector product (see "Hessian" on page 9-41).
`InitBarrierParam`	Initial barrier value. Sometimes it might help to try a value above the default 0.1, especially if the objective or constraint functions are large.
`InitTrustRegionRadius`	Initial radius of the trust region. On badly scaled problems it might help to choose a value smaller than the default $\sqrt{n}$, where n is the number of variables.
`MaxProjCGIter`	A tolerance (stopping criterion) for the number of projected conjugate gradient iterations; this is an inner iteration, not the number of iterations of the algorithm.
`ObjectiveLimit`	A tolerance (stopping criterion). If the objective function value goes below `ObjectiveLimit` and the iterate is feasible, the iterations halt, since the problem is presumably unbounded.
`ScaleProblem`	The default `obj-and-constr` causes the algorithm to normalize all constraints and the objective function. Disable by setting to `none`.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default `ldl-factorization` is usually faster than `cg` (conjugate gradient), though `cg` may be faster for large problems with dense Hessians.

`TolProjCG`	A relative tolerance (stopping criterion) for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.
`TolProjCGAbs`	Absolute tolerance (stopping criterion) for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.

Examples

Find values of x that minimize $f(x) = -x_1x_2x_3$, starting at the point `x = [10;10;10]`, subject to the constraints:

$$0 \leq x_1 + 2x_2 + 2x_3 \leq 72.$$

1. Write an M-file that returns a scalar value `f` of the objective function evaluated at `x`:

```
function f = myfun(x)
f = -x(1) * x(2) * x(3);
```

2. Rewrite the constraints as both less than or equal to a constant,

$$-x_1 - 2x_2 - 2x_3 \leq 0$$
$$x_1 + 2x_2 + 2x_3 \leq 72$$

3. Since both constraints are linear, formulate them as the matrix inequality $A \cdot x \leq b$, where

$$A = \begin{bmatrix} -1 & -2 & -2 \\ 1 & 2 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 72 \end{bmatrix}.$$

4. 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)
```

5 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

Trust-Region-Reflective Optimization

To use the trust-region-reflective algorithm, you must

- Supply the gradient of the objective function 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

You cannot use inequality constraints with the trust-region-reflective algorithm. If the preceding conditions are not met, `fmincon` reverts to the active-set algorithm.

`fmincon` returns a warning if you do not provide a gradient and the `Algorithm` option is `trust-region-reflective`. `fmincon` permits an approximate gradient to be supplied, but this option is not recommended; the numerical behavior of most optimization methods is considerably more robust when the true gradient is used.

The trust-region-reflective method in `fmincon` is in general 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, `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.

Take note of these characteristics of linearly constrained minimization:

- 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 $\overline{Aeq}$.

Active-Set Optimization

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, `'dependent'` appears 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'` appears under the `Procedures` heading.

Algorithm

Trust-Region-Reflective Optimization

The trust-region-reflective algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [3] and [4]. 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 “fmincon Trust Region Reflective Algorithm” on page 4-20.

Active-Set Optimization

`fmincon` uses a sequential quadratic programming (SQP) method. In this method, the function solves a quadratic programming (QP) subproblem at each iteration. `fmincon` updates an estimate of the Hessian of the Lagrangian at each iteration using the BFGS formula (see `fminunc` and references [7] and [8]).

`fmincon` performs a line search using a merit function similar to that proposed by [6], [7], and [8]. The QP subproblem is solved using an active set strategy similar to that described in [5]. “fmincon Active Set Algorithm” on page 4-26 describes this algorithm in detail.

See also “SQP Implementation” on page 4-29 for more details on the algorithm used.

Interior-Point Optimization

This algorithm is described in [1], [41], and [9].

Limitations

`fmincon` is a gradient-based method that is designed to work on problems where the objective and constraint functions are both continuous and have continuous first derivatives.

When the problem is infeasible, `fmincon` attempts to minimize the maximum constraint value.

The trust-region-reflective algorithm does not allow equal upper and lower bounds. For example, if `lb(2)==ub(2)`, `fmincon` gives this error:

```
Equal upper and lower bounds not permitted in this
large-scale method.
Use equality constraints and the medium-scale
method instead.
```

There are two different syntaxes for passing a Hessian, and there are two different syntaxes for passing a `HessMult` function; one for trust-region-reflective, and another for interior-point.

For trust-region-reflective, the Hessian of the Lagrangian is the same as the Hessian of the objective function. You pass that Hessian as the third output of the objective function.

For interior-point, the Hessian of the Lagrangian involves the Lagrange multipliers and the Hessians of the nonlinear constraint functions. You pass the Hessian as a separate function that takes into account both the position `x` and the Lagrange multiplier structure `lambda`.

Trust-Region-Reflective Coverage and Requirements

Additional Information Needed	For Large Problems
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.

References

[1] Byrd, R.H., J. C. Gilbert, and J. Nocedal, "A Trust Region Method Based on Interior Point Techniques for Nonlinear Programming," *Mathematical Programming*, Vol 89, No. 1, pp. 149–185, 2000.

[2] Byrd, R.H., Mary E. Hribar, and Jorge Nocedal, "An Interior Point Algorithm for Large-Scale Nonlinear Programming, SIAM Journal on Optimization," *SIAM Journal on Optimization*, Vol 9, No. 4, pp. 877–900, 1999.

[3] 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.

[4] 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.

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

[6] Han, S.P., "A Globally Convergent Method for Nonlinear Programming," Vol. 22, *Journal of Optimization Theory and Applications*, p. 297, 1977.

[7] 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.

[8] 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.

[9] Waltz, R. A., J. L. Morales, J. Nocedal, and D. Orban, "An interior algorithm for nonlinear optimization that combines line search and trust region steps," *Mathematical Programming*, Vol 107, No. 3, pp. 391–408, 2006.

See Also

`@ (function_handle)`, `fminbnd`, `fminsearch`, `fminunc`, `optimset`, `optimtool`

For more details about the `fmincon` algorithms, see "Constrained Nonlinear Optimization" on page 4-20. For more examples of nonlinear programming with constraints, see "Constrained Nonlinear Optimization Examples" on page 4-44.

Purpose

Solve minimax constraint problem

Equation

Finds the minimum of a problem specified by

$$\min_x \max_i F_i(x) \quad \text{such that} \quad \begin{cases} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub \end{cases}$$

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,x,A,b,Aeq,beq)
x = fminimax(fun,x,A,b,Aeq,beq,lb,ub)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fminimax(problem)
[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 (largest) value of a set of multivariable functions, starting at an initial estimate. This is generally referred to as the *minimax* problem.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective functions and nonlinear constraint functions, if necessary.

---

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 = fminimax(problem) finds the minimum for problem, where problem is a structure described in "Input Arguments" on page 9-61.

Create the structure problem by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

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

[x,fval,maxfval] = fminimax(...) returns the maximum of the objective functions in the input fun evaluated 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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fminimax`. This section provides function-specific details for `fun`, `nonlcon`, and `problem`:

`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);
```

If the user-defined values for `x` and `F` are matrices, they are converted to a vector using linear indexing.

To minimize the worst case absolute values of any of the elements of the vector $F(x)$ (i.e., $\min\{\max \text{abs}\{F(x)\}\}$), 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
```

Note Setting `GradObj` to `'on'` is effective only when there is no nonlinear constraint, or when the nonlinear constraint has `GradConstr` set to `'on'` as well. This is because internally the objective is folded into the constraints, so the solver needs both gradients (objective and constraint) supplied in order to avoid estimating a gradient.

`nonlcon` 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 `j`th column of `G` is the gradient of the `j`th objective function `F(j)`).

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)`. "Nonlinear Constraints" on page 2-14 explains how to "conditionalize" the gradients for use in solvers that do not accept supplied gradients, and explains the syntax of gradients.

Note Setting `GradConstr` to `'on'` is effective only when `GradObj` is set to `'on'` as well. This is because internally the objective is folded into the constraint, so the solver needs both gradients (objective and constraint) supplied in order to avoid estimating a gradient.

Note Because Optimization Toolbox functions only accept inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

"Passing Extra Parameters" on page 2-17 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

`problem`	`objective`	Objective function
	`x0`	Initial point for `x`
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`lb`	Vector of lower bounds
	`ub`	Vector of upper bounds
	`nonlcon`	Nonlinear constraint function
	`solver`	`'fminimax'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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:

`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.
	lssteplength	Size of line search step relative to search direction
	stepsize	Final displacement in x
	algorithm	Optimization algorithm used.
	firstorderopt	Measure of first-order optimality
	constrviolation	Maximum of nonlinear constraint functions
	message	Exit message

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 7-7 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; `'notify'` displays output only if the function does not converge;`'final'` (default) displays just the final output.
`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations but should be more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. So, for example, it may take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
`FunValCheck`	Check whether objective function and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is `complex`, `Inf`, or `NaN`. `'off'` displays no error.

GradConstr	Gradient for the user-defined constraints. See the preceding description of nonlcon to see how to define the gradient in nonlcon.
GradObj	Gradient for the user-defined objective function. See the preceding description of fun to see how to define the gradient in fun.
MaxFunEvals	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations allowed.
MaxSQPIter	Maximum number of SQP 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 one or more user-defined functions that are called after each iteration of an optimization (medium scale algorithm only). See "Output Function" on page 7-17.

PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying @optimplotx plots the current point; @optimplotfunccount plots the function count; @optimplotfval plots the function value; @optimplotconstrviolation plots the maximum constraint violation; @optimplotstepsize plots the step size; @optimplotfirstorderopt plots the first-order of optimality.
RelLineSrchBnd	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in x satisfies $\|\Delta x(i)\| \leq \text{relLineSrchBnd} \cdot \max(\|x(i)\|, \|typicalx(i)\|)$. This option provides control over the magnitude of the displacements in x for cases in which the solver takes steps that are considered too large.
RelLineSrchBndDuration	Number of iterations for which the bound specified in RelLineSrchBnd should be active (default is 1).
TolCon	Termination tolerance on the constraint violation.
TolConSQP	Termination tolerance on inner iteration SQP constraint violation.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.

TypicalX	Typical x values. The length of the vector is equal to the number of elements in x0, the starting point.
UseParallel	When 'always', estimate gradients in parallel. Disable by setting to 'never'.

Examples

Find values of x that minimize the maximum value of

$$[f_1(x), f_2(x), f_3(x), f_4(x), f_5(x)]$$

where

$$f_1(x) = 2x_1^2 + x_2^2 - 48x_1 - 40x_2 + 304,$$
$$f_2(x) = -x_1^2 - 3x_2^2,$$
$$f_3(x) = x_1 + 3x_2 - 18,$$
$$f_4(x) = -x_1 - x_2,$$
$$f_5(x) = x_1 + x_2 - 8.$$

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

$$[f_1(x), f_2(x), f_3(x), f_4(x), f_5(x)]$$

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 abs. 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 internally reformulates the minimax problem into an equivalent Nonlinear Linear Programming problem by appending additional (reformulation) constraints of the form $F_i(x) \leq \gamma$ to the constraints given in "Equation" on page 9-59, and then minimizing γ over x. fminimax uses a sequential quadratic programming (SQP) method [1] to solve this problem.

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 4-29 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.

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 Nonlinearly Constrained Optimization Calculations,” *Numerical Analysis*, ed. G.A. Watson, *Lecture Notes in Mathematics*, Vol. 630, Springer Verlag, 1978.

See Also

`@ (function_handle)`, `fgoalattain`, `lsqnonlin`, `optimset`, `optimtool`

For more details about the `fminimax` algorithm, see “Multiobjective Optimization” on page 4-141. For another example of minimizing the maximum, see “Multiobjective Optimization Examples” on page 4-147.

Purpose

Find minimum of unconstrained multivariable function using derivative-free method

Equation

Finds the minimum of a problem specified by

$$\min_x f(x)$$

where x is a vector and $f(x)$ is a function that returns a scalar.

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
x = fminsearch(problem)
[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*.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective function, if necessary.

`x = fminsearch(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`. `fun` is a function handle for either an M-file function or an anonymous function. `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 = fminsearch(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-74.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fminsearch`. This section provides function-specific details for `fun`, `options`, and `problem`:

`fun` — The function to be minimized. `fun` is a function handle for 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)
```

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, such as

```
x = fminsearch(@(x)norm(x)^2,x0,A,b);
```

`options` — "Options" on page 9-76 provides the function-specific details for the `options` values.

problem	objective	Objective function
	x0	Initial point for x
	solver	'fminsearch'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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	
	iterations	Number of iterations
	funcCount	Number of function evaluations
	algorithm	Optimization algorithm used
	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 7-7 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 and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is `complex` or `NaN`. `'off'` (the default) displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx` plots the current point; `@optimplotfunccount` plots the function count; `@optimplotfval` plots the function value.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.

Examples

Example 1

A classic test example for multidimensional minimization is the Rosenbrock `banana` function

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

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The anonymous function shown here defines the function and returns a function handle called `banana`:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval,exitflag] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =
    1.0000    1.0000

fval =
    8.1777e-010

exitflag =
     1
```

This indicates that the minimizer was found at [1 1] with a value near zero.

Example 2

You can modify the first example by adding a parameter a to the second term of the banana function:

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

This changes the location of the minimum to the point `[a,a^2]`. To minimize this function for a specific value of `a`, for example a = `sqrt(2)`, create a one-argument anonymous function that captures the value of `a`.

```
a = sqrt(2);
banana = @(x)100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval,exitflag] = fminsearch(banana, [-1.2, 1], ...
   optimset('TolX',1e-8));
```

seeks the minimum `[sqrt(2), 2]` to an accuracy higher than the default on `x`. The result is

```
x =
    1.4142    2.0000

fval =
  4.2065e-018

exitflag =
     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 solves nondifferentiable problems and 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.

Notes

fminsearch is not the preferred choice for solving problems that are sums of squares, that is, of the form

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$

Instead use the lsqnonlin function, which has been optimized for problems of this form.

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.

See Also

@ (function_handle), fminbnd, fminunc, optimset, optimtool, "Anonymous Functions"

fminunc

Purpose

Find minimum of unconstrained multivariable function

Equation

Finds the minimum of a problem specified by

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

where x is a vector and $f(x)$ is a function that returns a scalar.

Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(problem)
[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*.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective function, if necessary.

`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 = fminunc(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-81.

Create the structure problem by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

[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 9-84.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into fminunc. This section provides function-specific details for fun, options, and problem:

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. The gradient is the partial derivatives $\partial f/\partial x_i$ 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. 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.

"Writing Objective Functions" on page 2-4 explains how to "conditionalize" the gradients and Hessians for use in solvers that do not accept them. "Passing Extra Parameters" on page 2-17 explains how to parameterize fun, if necessary.

options	"Options" on page 9-84 provides the function-specific details for the options values.	
problem	objective	Objective function
	x0	Initial point for x
	solver	'fminunc'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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` — Magnitude of gradient smaller than the specified tolerance.
	`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 cannot find an acceptable point 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
	`firstorderopt` — Measure of first-order optimality
	`algorithm` — Optimization algorithm used

cgiterations	Total number of PCG iterations (large-scale algorithm only)
stepsize	Final displacement in x (medium-scale algorithm only)
message	Exit message

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 7-7 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.
`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'notify'` displays output only if the function does not converge;`'final'` (default) displays just the final output.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function return a value that is `complex` or `NaN`. `'off'` (the default) displays no error.
`GradObj`	Gradient for the objective function that you define. See the preceding description of `fun` to see how to define the gradient in `fun`.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.

OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying @optimplotx plots the current point; @optimplotfunccount plots the function count; @optimplotfval plots the function value; @optimplotstepsize plots the step size; @optimplotfirstorderopt plots the first-order of optimality.
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)` where Hinfo contains the matrix 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. See "Passing Extra Parameters" on page 2-17 for information on how to supply values for any additional parameters hmfun needs.

Note 'Hessian' must be set to 'on' for Hinfo to be passed from fun to hmfun.

See "Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64 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 9-91).
`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. Setting `PrecondBandWidth` to `'Inf'` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.
`TolPCG`	Termination tolerance on the PCG iteration.

Medium-Scale Algorithm Only

These options are used only by the medium-scale algorithm:

`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations but should be more accurate.
`HessUpdate`	Method for choosing the search direction in the Quasi-Newton algorithm. The choices are • `'bfgs'` • `'dfp'` • `'steepdesc'` See "Hessian Update" on page 4-11 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-006 *
```

```
    0.2541   -0.2029

fval =
  1.3173e-013
```

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 *
    0.3331   -0.4441

fval =
  2.3419e-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

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$

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 you 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] and [3]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See “Large Scale fminunc Algorithm” on page 4-3, “Trust-Region Methods for Nonlinear Minimization” on page 4-3 and “Preconditioned Conjugate Gradient Method” on page 4-23.

Medium-Scale Optimization

`fminunc`, with the `LargeScale` option set to `'off'` with `optimset`, uses the BFGS Quasi-Newton method with a cubic line search procedure. This quasi-Newton method uses the BFGS ([1],[5],[8], and [9]) formula for updating the approximation of the Hessian matrix. You can select the DFP ([4],[6], and [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.

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, you must supply the gradient in fun (and GradObj must be set 'on' in options).

Large-Scale Problem Coverage and Requirements

Additional Information Needed	For Large Problems
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.

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.

See Also

`@ (function_handle)`, `fminsearch`, `optimset`, `optimtool`, "Anonymous Functions"

For more details about the `fminunc` algorithms, see "Unconstrained Nonlinear Optimization" on page 4-3. For more example of unconstrained nonlinear programming, see "Unconstrained Nonlinear Optimization Examples" on page 4-14.

Purpose

Find minimum of semi-infinitely constrained multivariable nonlinear function

Equation

Finds the minimum of a problem specified by

$$\min_x f(x) \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub, \\ c(x) \le 0, \\ ceq(x) = 0, \\ K_i(x, w_i) \le 0, \ 1 \le i \le n. \end{cases}$$

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 $w_1,w_2,...,w_n$. The variables $w_1,w_2,...,w_n$ are vectors of, at most, length two.

Syntax

```
x = fseminf(fun,x0,ntheta,seminfcon)
x = fseminf(fun,x0,ntheta,seminfcon,A,b)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub)
x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub,
   options)
x = fseminf(problem)
[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 $w_i \Re^1$ (or $w_i \Re^2$). 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.

Note “Passing Extra Parameters” on page 2-17 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`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 = fseminf(problem)` finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 9-96.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in “Exporting to the MATLAB Workspace” on page 3-14.

`[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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fseminf`. This section provides function-specific details for `fun`, `ntheta`, `options`, `seminfcon`, and `problem`:

`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`.

`ntheta` — The number of semi-infinite constraints.

`options` — "Options" on page 9-100 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 "Example: Two-Dimensional Semi-Infinite Constraint" on page 4-70). On the first iteration `S` is `NaN`, so that some initial sampling interval must be determined by `seminfcon`.

Note Because Optimization Toolbox functions only accept inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

"Passing Extra Parameters" on page 2-17 explains how to parameterize `seminfcon`, if necessary. "Example of Creating Sampling Points" on page 4-41 contains an example of both one- and two-dimensional sampling points.

`problem`	`objective`	Objective function
	`x0`	Initial point for `x`
	`ntheta`	Number of semi-infinite constraints
	`seminfcon`	Semi-infinite constraint function
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`lb`	Vector of lower bounds
	`ub`	Vector of upper bounds

`solver`	`'fseminf'`
`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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
`lssteplength`	Size of line search step relative to search direction
`stepsize`	Final displacement in `x`
`algorithm`	Optimization algorithm used
`constrviolation`	Maximum of nonlinear constraint functions
`firstorderopt`	Measure of first-order optimality
`message`	Exit message

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 7-7 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; `'notify'` displays output only if the function does not converge;`'final'` (default) displays just the final output.
`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations but should be more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. So, for example, it may take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
`FunValCheck`	Check whether objective function and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is `complex`, `Inf`, or `NaN`. `'off'` (the default) displays no error.
`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`.
`MaxFunEvals`	Maximum number of function evaluations allowed
`MaxIter`	Maximum number of iterations allowed

`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx` plots the current point; `@optimplotfunccount` plots the function count; `@optimplotfval` plots the function value; `@optimplotconstrviolation` plots the maximum constraint violation; `@optimplotstepsize` plots the step size.
`RelLineSrchBnd`	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in `x` satisfies $\lvert\Delta x(i)\rvert \leq \text{relLineSrchBnd} \cdot \max(\lvert x(i)\rvert, \lvert typicalx(i)\rvert)$. This option provides control over the magnitude of the displacements in `x` for cases in which the solver takes steps that are considered too large.
`RelLineSrchBndDuration`	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is 1)
`TolCon`	Termination tolerance on the constraint violation
`TolConSQP`	Termination tolerance on inner iteration SQP 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 $K_i(x,w_i)$.

Example

This example minimizes the function

$$(x - 1)^2,$$

subject to the constraints

$$0 \le x \le 2$$
$$g(x,\ t) = (x - 1/2) - (t - 1/2)^2 \le 0 \text{ for all } 0 \le t \le 1.$$

The unconstrained objective function is minimized at $x = 1$. However, the constraint,

$$g(x,\ t) \le 0 \text{ for all } 0 \le t \le 1,$$

implies $x \le 1/2$. You can see this by noticing that $(t - 1/2)^2 \ge 0$, so

$$\max_t\ g(x,\ t) = (x - 1/2).$$

Therefore

$$\max_t\ g(x,\ t) \le 0 \text{ when } x \le 1/2.$$

To solve this problem using `fseminf`:

1. Write the objective function as an anonymous function:

```
objfun = @(x)(x-1)^2;
```

2. Write the semi-infinite constraint function, which includes the nonlinear constraints ([ ] in this case), initial sampling interval for t (0 to 1 in steps of 0.01 in this case), and the semi-infinite constraint function $g(x, t)$:

```
function [c, ceq, K1, s] = seminfcon(x,s)

% No finite nonlinear inequality and equality constraints
c = [];
ceq = [];

% Sample set
if isnan(s)
    % Initial sampling interval
    s = [0.01 0];
end
t = 0:s(1):1;

% Evaluate the semi-infinite constraint
K1 = (x - 0.5) - (t - 0.5).^2;
```

3. Call `fseminf` with initial point 0.2, and view the result:

```
x = fseminf(objfun,0.2,1,@seminfcon)

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 =
    0.5000
```

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 `fmincon` function. 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.

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`, see also “SQP Implementation” on page 4-29. For more details on the `fseminf` algorithm, see “fseminf Problem Formulation and Algorithm” on page 4-39.

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`, `optimtool`

For more details about the `fseminf` algorithm, see “fseminf Problem Formulation and Algorithm” on page 4-39. For more examples of semi-infinitely constrained problems, see “Constrained Nonlinear Optimization Examples” on page 4-44.

fsolve

Purpose

Solve system of nonlinear equations

Equation

Solves a problem specified by

$$F(x) = 0$$

for x, where x is a vector and $F(x)$ is a function that returns a vector value.

Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
x = fsolve(problem)
[x,fval] = fsolve(fun,x0)
[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.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the system of equations, if necessary.

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations described in `fun`.

`x = fsolve(fun,x0,options)` solves the equations with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fsolve(problem)` solves `problem`, where `problem` is a structure described in "Input Arguments" on page 9-107.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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`.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fsolve`. This section provides function-specific details for `fun` and `problem`:

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 user-defined values for `x` and `F` are matrices, they are converted to a vector using linear indexing.

If the Jacobian can also be computed *and* the Jacobian option is `'on'`, set by

```
options = optimset('Jacobian','on')
```

the function `fun` must return, in a second output argument, the Jacobian value `J`, a matrix, at `x`.

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, where `n` is the length of `x0`, 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)`. (The Jacobian `J` is the transpose of the gradient of `F`.)

`problem`	`objective`	Objective function
	`x0`	Initial point for `x`
	`solver`	`'fsolve'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments returned by `fsolve`. For more information on the output headings for `fsolve`, see "Function-Specific Output Headings" on page 2-48.

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`	Output function terminated the algorithm.
`-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`	Optimization algorithm used.
`cgiterations`	Total number of PCG iterations (large-scale algorithm only)
`stepsize`	Final displacement in `x` (Gauss-Newton and Levenberg-Marquardt algorithms)
`firstorderopt`	Measure of first-order optimality (dogleg or large-scale algorithm, [ ] for others)
`message`	Exit message

Options

Optimization options used by `fsolve`. Some options apply to all algorithms, some are only relevant when using the trust-region-reflective algorithm, and others are only relevant when using the other algorithms. You can use `optimset` to set or change the values of these fields in the options structure, `options`. See "Optimization Options" on page 7-7 for detailed information.

The `Algorithm` option specifies a preference for which algorithm to use. It is only a preference because for the trust-region-reflective 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` or else the default trust-region-dogleg algorithm is used.

All Algorithms

These options are used by all algorithms:

`Algorithm`	Choose between `'trust-region-dogleg'`, `'trust-region-reflective'`, and `'levenberg-marquardt'`.
`DerivativeCheck`	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be solved.
`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. `'off'` (the default) displays no error.
`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.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx` plots the current point; `@optimplotfunccount` plots the function count; `@optimplotfval` plots the function value; `@optimplotresnorm` plots the norm of the residuals; `@optimplotstepsize` plots the step size; `@optimplotfirstorderopt` plots the first-order of optimality.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Trust-Region-Reflective 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)
```

where `Jinfo` contains a matrix 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`, `W = J'*(J*Y)`.
- If `flag > 0`, `W = J*Y`.
- If `flag < 0`, `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. See "Passing Extra Parameters" on page 2-17 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64 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 9-117).
PrecondBandWidth	The default PrecondBandWidth is 'Inf', which means a direct factorization (Cholesky) is used rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution. Set PrecondBandWidth to 0 for diagonal preconditioning (upper bandwidth of 0). For some problems, an intermediate bandwidth reduces the number of PCG iterations.
TolPCG	Termination tolerance on the PCG iteration.

Levenberg-Marquardt Algorithm Only

This option is used only by the Levenberg-Marquardt algorithm:

ScaleProblem	'Jacobian' can sometimes improve the convergence of a poorly scaled problem; the default is 'none'.

Gauss-Newton Algorithm Only

These options are used only by the Gauss-Newton algorithm:

LargeScale and NonlEqnAlgorithm	Specify LargeScale as 'off' and NonlEqnAlgorithm as 'gn' to choose the Gauss-Newton algorithm. These options are being obsoleted, and are not used for the other algorithms.
LineSearchType	The choices are 'cubicpoly' or the default 'quadcubic'.

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

$$\begin{aligned} 2x_1 - x_2 - e^{-x_1} &= 0 \\ -x_1 + 2x_2 - e^{-x_2} &= 0, \end{aligned}$$

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`\` (matrix left division) for better speed and accuracy. For example, to find the solution to the following linear system of equations:

$$3x_1 + 11x_2 - 2x_3 = 7$$
$$x_1 + x_2 - 2x_3 = 4$$
$$x_1 - x_2 + x_3 = 19.$$

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 trust-region-reflective 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” on page 9-119 and “Diagnostics” on page 9-118 following).

- By default `fsolve` chooses the trust-region dogleg algorithm. The algorithm is a variant of the Powell dogleg method described in [8]. It is similar in nature to the algorithm implemented in [7]. It is described in the User’s Guide in “Trust-Region Dogleg Method” on page 4-155.
- The trust-region-reflective algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [1] and [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned

conjugate gradients (PCG). See “Trust-Region Reflective fsolve Algorithm” on page 4-157.

- The Levenberg-Marquardt method is described in [4], [5], and [6]. It is described in the User’s Guide in “Levenberg-Marquardt Method” on page 4-160.
- The medium-scale Gauss-Newton method [3] with line search is described in the User’s Guide in “Gauss-Newton Method” on page 4-160. The default line search algorithm for the Gauss-Newton algorithm , i.e., the `LineSearchType` option, is `'quadcubic'`. This 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.

Diagnostics

All Algorithms

`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.

Trust-Region-Dogleg Algorithm

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.

Trust-Region-Reflective Algorithm

The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective algorithm 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.

Large-Scale Problem Coverage and Requirements

For Large Problems
• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.

Number of Equations

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.

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.

See Also

`@ (function_handle)`, `\` (matrix left division), `lsqcurvefit`, `lsqnonlin`, `optimset`, `optimtool`, "Anonymous Functions"

For more details about the `fsolve` algorithms, see "Equation Solving" on page 4-154. For more examples of equation solving, see "Equation Solving Examples" on page 4-162.

Purpose

Find root of continuous function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
x = fzero(problem)
[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. `fun` is a function handle for either an M-file function or an anonymous function. 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.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to `fun`, if necessary.

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. An interval `x0` must be finite; it cannot contain ±`Inf`.

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)` solves the equation with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fzero(problem)` solves `problem`, where `problem` is a structure described in "Input Arguments" on page 9-122.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `fzero`. This section provides function-specific details for `fun`, `options`, and `problem`:

`fun` The function whose zero is to be computed. `fun` is a function handle for a function that accepts a scalar `x` and returns a scalar, 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 an error when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no error.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`TolX`	Termination tolerance on `x`.

`problem`	`objective`	Objective function
	`x0`	Initial point for `x`, scalar or 2–dimensional vector
	`solver`	`'fzero'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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	
	`intervaliterations`	Number of iterations taken to find an interval containing a root
	`iterations`	Number of zero-finding iterations
	`funcCount`	Number of function evaluations

`algorithm`	Optimization algorithm used
`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
```

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.

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.

See Also

`@ (function_handle)`, `\` (matrix left division), `fminbnd`, `fsolve`, `optimset`, `optimtool`, `roots`, “Anonymous Functions”

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` times the maximum absolute value in that column.

See Also `sprank`, `spy`

Purpose

Find minimum of constrained or unconstrained nonlinear multivariable function using KNITRO third-party libraries

Equation

Finds the minimum of a problem specified by

$$\min_{x} f(x) \text{ such that } \begin{cases} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub, \end{cases}$$

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. All constraints are optional; `ktrlink` can minimize unconstrained problems.

Syntax

```
x = ktrlink(fun,x0)
x = ktrlink(fun,x0,A,b)
x = ktrlink(fun,x0,A,b,Aeq,beq)
x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub)
x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,
knitroOptions)
[x,fval] = ktrlink(...)
[x,fval,exitflag] = ktrlink(...)
[x,fval,exitflag,output] = ktrlink(...)
[x,fval,exitflag,output,lambda] = ktrlink(...)
```

Description

`ktrlink` attempts to find a minimum of a scalar function of several variables starting at an initial estimate. This is generally referred to as *constrained* or *unconstrained nonlinear optimization*, or *nonlinear programming*.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`x = ktrlink(fun,x0)` starts at `x0` and attempts to find a minimizer `x` of the function described in `fun`, subject to no constraints. `x0` can be a scalar, vector, or matrix.

`x = ktrlink(fun,x0,A,b)` minimizes `fun` subject to the linear inequalities `A*x` ≤ `b`.

`x = ktrlink(fun,x0,A,b,Aeq,beq)` minimizes `fun` subject to the linear equalities `Aeq*x = beq` as well as `A*x` ≤ `b`. If no inequalities exist, set `A = []` and `b = []`.

`x = ktrlink(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`. If no equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` is unbounded below, set `lb(i) = -Inf`, and if `x(i)` is unbounded above, set `ub(i) = Inf`.

`x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimization to the nonlinear inequalities `c(x)` and the equalities `ceq(x)` defined in `nonlcon`. `fmincon` optimizes such that `c(x)` ≤ `0` and `ceq(x) = 0`. If no bounds exist, set `lb = []` and/or `ub = []`.

`x = ktrlink(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. If there are no nonlinear inequality or equality constraints, set `nonlcon = []`.

`x = ktrlink(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options, knitroOptions)` minimizes with the KNITRO options specified in the text file `knitroOptions`. All options given in `options` are ignored except for `HessFcn`, `HessMult`, `HessPattern`, and `JacobPattern`.

`[x,fval] = ktrlink(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = ktrlink(...)` returns `exitflag`, which describes the exit condition of the KNITRO solver.

`[x,fval,exitflag,output] = ktrlink(...)` returns a structure `output` with information about the optimization.

`[x,fval,exitflag,output,lambda] = ktrlink(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Components of `x0` that violate the bounds `lb ≤ x ≤ ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

Input Arguments

"Function Arguments" on page 7-2 contains descriptions of arguments passed to `ktrlink`. "Options" on page 9-44 provides the function-specific details for the `options` values. This section provides function-specific details for `fun` and `nonlcon`.

`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`. `fun` can be specified as a function handle for an M-file function

```
x = ktrlink(@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 = ktrlink(@(x)norm(x)^2,x0,A,b);
```

If you can compute the gradient of `fun` *and* the `GradObj` option is `'on'`, as set by

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

then `fun` must return the gradient vector `g(x)` in the second output argument.

If you can compute the Hessian matrix, there are several ways to pass the Hessian to `ktrlink`. See “Hessian” on page 9-134 for details.

`nonlcon` The function that computes the nonlinear inequality constraints `c(x)≤ 0` and the nonlinear equality constraints `ceq(x) = 0`. `nonlcon` accepts a vector `x` and returns the two vectors `c` and `ceq`. `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 = ktrlink(@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 you can compute the gradients of the constraints *and* the `GradConstr` option is `'on'`, as set by

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

then `nonlcon` must also return `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`, in the third and fourth output arguments respectively. See “Nonlinear Constraints” on page 2-14 for details.

Note Because Optimization Toolbox functions only accept inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

"Passing Extra Parameters" on page 2-17 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

Output Arguments

"Function Arguments" on page 7-2 contains descriptions of arguments returned by `ktrlink`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag`	Integer identifying the reason the algorithm terminated. For more information, see the KNITRO documentation at `http://www.ziena.com/`	
`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

`constrviolation`	Maximum of constraint violations (interior-point algorithm only)
`firstorderopt`	Measure of first-order optimality

Hessian

`ktrlink` can optionally use a user-supplied Hessian, the matrix of second derivatives of the Lagrangian, namely,

$$\nabla_{xx}^2 L(x,\lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x). \qquad \textbf{(9-2)}$$

If you don't supply a Hessian, KNITRO software estimates it.

To provide a Hessian, the syntax is

```
hessian = hessianfcn(x, lambda)
```

`hessian` is an n-by-n matrix, sparse or dense, where n is the number of variables. `lambda` is a structure with the Lagrange multiplier vectors associated with the nonlinear constraints:

```
lambda.ineqnonlin
lambda.eqnonlin
```

KNITRO software computes `lambda`. `hessianfcn` must calculate the sums in Equation 9-2. Indicate that you are supplying a Hessian by

```
options = optimset('Hessian','user-supplied',...
                   'HessFcn',@hessianfcn);
```

There are several more options for Hessians:

- `options = optimset('Hessian','bfgs');`

  The KNITRO solver calculates the Hessian by a dense quasi-Newton approximation.

- `options = optimset('Hessian',...`
  `{'lbfgs',positive integer});`

  The KNITRO solver calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation. The positive integer specifies how many past iterations should be remembered.

- `options = optimset('Hessian','fin-diff-grads',...`
  `'SubproblemAlgorithm','cg','GradObj','on',...`
  `'GradConstr','on');`

  The KNITRO solver calculates a Hessian-times-vector product by finite differences of the gradient(s). You must supply the gradient of the objective function, and also gradients of any nonlinear constraint functions.

- `options = optimset('Hessian','user-supplied',...`
  `'SubproblemAlgorithm','cg','HessMult',@HessMultFcn]);`

  The KNITRO solver uses a Hessian-times-vector product. You must supply the function `HessMultFcn`, which returns an *n*-by-1 vector. The `HessMult` option enables you to pass the result of multiplying the Hessian by a vector without calculating the Hessian.

The syntax for the `'HessMult'` option is:

```
w = HessMultFcn(x,lambda,v);
```

The result `w` should be the product `H*v`, where `H` is the Hessian at `x`, `lambda` is the Lagrange multiplier (computed by KNITRO software), and `v` is a vector.

Options

Optimization options used by `ktrlink`. Use `optimset` to set or change the values of fields in the options structure `options`. See “Optimization Options” on page 7-7 for detailed information. For example:

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

Option	Description
Algorithm	Choose a KNITRO optimization algorithm: 'interior-point' or 'active-set'.
AlwaysHonorConstraints	The default 'bounds' ensures that bound constraints are satisfied at every iteration. Disable by setting to 'none'.
Display	Level of display: • 'off' displays no output. • 'iter' displays output at each iteration. • 'final' (default) displays just the final output.
FinDiffType	Finite differences, used to estimate gradients, are either 'forward' (the default), or 'central' (centered). 'central' takes twice as many function evaluations but should be more accurate. 'central' differences might violate bounds during their evaluation.
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. It is optional for the 'active-set' and 'interior-point' methods.
HessFcn	Function handle to a user-supplied Hessian (see "Hessian" on page 9-41).
Hessian	Chooses how ktrlink calculates the Hessian (see "Hessian" on page 9-41).
HessMult	Handle to a user-supplied function that gives a Hessian-times-vector product (see "Hessian" on page 9-41).

Option	Description
`InitBarrierParam`	Initial barrier value. A value above the default 0.1 might help, especially if the objective or constraint functions are large.
`InitTrustRegionRadius`	Initial radius of the trust region. On badly-scaled problems choose a value smaller than the default, $\sqrt{n}$, where n is the number of variables.
`MaxIter`	Maximum number of iterations allowed.
`MaxProjCGIter`	A tolerance (stopping criterion) for the number of projected conjugate gradient iterations; this is an inner iteration, not the number of iterations of the algorithm.
`ObjectiveLimit`	A tolerance (stopping criterion). If the objective function value goes below `ObjectiveLimit` and the iterate is feasible, the iterations halt, since the problem is presumably unbounded.
`ScaleProblem`	The default `'obj-and-constr'` causes the algorithm to normalize all constraints and the objective function. Disable by setting to `'none'`.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default `'ldl-factorization'` is usually faster than `'cg'` (conjugate gradient), though `'cg'` may be faster for large problems with dense Hessians.
`TolFun`	Termination tolerance on the function value.

Option	Description
TolCon	Termination tolerance on the constraint violation.
TolX	Termination tolerance on x.

KNITRO Options

You can set options for the KNITRO libraries and pass them in a text file. The text file should consist of lines of text with the name of an option followed by blank space and then the desired value of the option. For example, to select the maximum run time to be less than 100 seconds, and to use an adaptive algorithm for changing the multiplier μ, create a text file containing the following lines:

```
ms_maxtime_real 100
bar_murule adaptive
```

For full details about the structure of the file and all possible options, see the KNITRO documentation at http://www.ziena.com/.

References

[1] http://www.ziena.com/

See Also

@ (function_handle), fminbnd, fmincon, fminsearch, fminunc, optimset

linprog

Purpose

Solve linear programming problems

Equation

Finds the minimum of a problem specified by

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

f, x, b, beq, lb, and ub are vectors, and A and Aeq are matrices.

Syntax

```
x = linprog(f,A,b)
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 = linprog(problem)
[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 = linprog(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-140.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[x,fval] = linprog(...)` returns the value of the objective function `fun` at the solution `x`: `fval = f'*x`.

`[x,fval,exitflag] = linprog(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `linprog`. "Options" on page 9-142 provides the function-specific details for the `options` values.

`problem`	`f`	Linear objective function vector `f`
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`lb`	Vector of lower bounds
	`ub`	Vector of upper bounds
	`x0`	Initial point for `x`, active set algorithm only
	`solver`	`'linprog'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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:	
	iterations	Number of iterations
	algorithm	Optimization algorithm used
	cgiterations	0 (large-scale algorithm only, included for backward compatibility)
	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 7-7 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 and simplex algorithms.
`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 "Medium-Scale linprog Simplex Algorithm" on page 4-82 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

$$x_1 - x_2 + x_3 \le 20$$
$$3x_1 + 2x_2 + 4x_3 \le 42$$
$$3x_1 + 2x_2 \le 30$$
$$0 \le x_1,\ 0 \le x_2,\ 0 \le x_3.$$

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 4-74.

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 "Medium-Scale linprog Simplex Algorithm" on page 4-82, 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 4-74). 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-side 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 4-74 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.

Large-Scale Optimization

Large-Scale Problem Coverage and Requirements

For Large Problems
A and *Aeq* should be sparse.

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.

See Also

`quadprog`, `optimtool`

For more details about the `linprog` algorithms, see "Linear Programming" on page 4-74. For more examples of linear programming, see "Linear Programming Examples" on page 4-86.

Purpose

Solve nonlinear curve-fitting (data-fitting) problems in least-squares sense

Equation

Find coefficients x that best fit the equation

$$\min_x \|F(x, xdata) - ydata\|_2^2 = \min_x \sum_i \left(F(x_i, xdata_i) - ydata_i\right)^2,$$

given input data *xdata,* and the observed output *ydata*, where *xdata* and *ydata* are matrices or vectors of length m, and $F(x, xdata)$ is a matrix-valued or vector-valued function.

The `lsqcurvefit` function uses the same algorithm as `lsqnonlin`, and provides an interface designed specifically for data-fitting problems.

Syntax

```
x = lsqcurvefit(fun,x0,xdata,ydata)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)
x = lsqcurvefit(problem)
[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`.

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the function F, if necessary.

`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 = lsqcurvefit(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-152.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.

Components of `x0` that violate the bounds `lb` ≤ `x` ≤ `ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `lsqcurvefit`. This section provides function-specific details for `fun`, `options`, and `problem`:

`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);
```

If the user-defined values for `x` and `F` are matrices, they are converted to a vector using linear indexing.

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`. 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)`. (The Jacobian `J` is the transpose of the gradient of `F`.) For more information, see "Jacobians of Vector and Matrix Objective Functions" on page 2-6.

`options` "Options" on page 9-156 provides the function-specific details for the `options` values.

problem	objective	Objective function of x and xdata
	x0	Initial point for x, active set algorithm only
	xdata	Input data for objective function
	ydata	Output data to be matched by objective function
	lb	Vector of lower bounds
	ub	Vector of upper bounds
	solver	'lsqcurvefit'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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`	Output function terminated the algorithm.
	`-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	
	`firstorderopt`	Measure of first-order optimality (trust-region-reflective algorithm, [ ] for others).
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`cgiterations`	Total number of PCG iterations (trust-region-reflective algorithm, [ ] for others)
	`algorithm`	Optimization algorithm used
	`stepsize`	Final displacement in `x` (Levenberg-Marquardt algorithm).
	`message`	Exit message

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 trust-region-reflective algorithm, and others are only relevant when you are using the Levenberg-Marquardt algorithm. Use `optimset` to set or change the values of these fields in the options structure `options`. See “Optimization Options” on page 7-7 for detailed information.

The `Algorithm` option specifies a preference for which algorithm to use. It is only a preference, because certain conditions must be met to use the trust-region-reflective or Levenberg-Marquardt algorithm. For the trust-region-reflective 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 trust-region-reflective algorithm handles bound constraints:

`Algorithm`	Use trust-region-reflective algorithm if possible when set to `'trust-region-reflective'`. Use Levenberg-Marquardt algorithm when set to `'levenberg-marquardt'`.

Algorithm Options

These options are used by both algorithms:

`Algorithm`	Use trust-region-reflective algorithm if possible when set to `'trust-region-reflective'`. Use Levenberg-Marquardt algorithm when set to `'levenberg-marquardt'`. Set the Levenberg-Marquardt parameter λ by setting `Algorithm` to a cell array such as `{'levenberg-marquardt',.005}`. Default $\lambda = 0.01$.
`DerivativeCheck`	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`Display`	Level of display. `'off'` displays no output, and `'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 one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.

PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots, or write your own. Specifying @optimplotx plots the current point; @optimplotfunccount plots the function count; @optimplotfval plots the function value.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.
TypicalX	Typical x values.

Trust-Region-Reflective Algorithm Only

These options are used only by the trust-region-reflective 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)
```

where Jinfo contains the matrix 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. `lsqcurvefit` uses `Jinfo` to compute the preconditioner. See "Passing Extra Parameters" on page 2-17 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See "Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64 and "Example: Jacobian Multiply Function with Linear Least Squares" on page 4-134 for similar examples.

`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 9-162).
PrecondBandWidth	The default PrecondBandWidth is 'Inf', which means a direct factorization (Cholesky) is used rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution. Set PrecondBandWidth to 0 for diagonal preconditioning (upper bandwidth of 0). For some problems, an intermediate bandwidth reduces the number of PCG iterations.
TolPCG	Termination tolerance on the PCG iteration.

Levenberg-Marquardt Algorithm Only

This option is used only by the Levenberg-Marquardt algorithm:

ScaleProblem	'Jacobian' can sometimes improve the convergence of a poorly-scaled problem; the default is 'none'.

Gauss-Newton Algorithm Only

These options are used only by the Gauss-Newton algorithm:

LargeScale and LevenbergMarquardt	Specify LargeScale as 'off' and LevenbergMarquardt as 'off' to choose the Gauss-Newton algorithm. These options are being obsoleted, and are not used for the other algorithms.
LineSearchType	The choices are 'cubicpoly' or the default 'quadcubic'.

Examples

Given vectors of data *xdata* and *ydata*, suppose you want to find coefficients *x* to find the best fit to the exponential decay equation

$$ydata(i) = x(1)e^{x(2)xdata(i)}$$

That is, you want to minimize

$$\min_x \sum_i \left(F\left(x, xdata_i\right) - ydata_i\right)^2,$$

where m is the length of `xdata` and `ydata`, the function `F` is defined by

```
F(x,xdata) = x(1)*exp(x(2)*xdata);
```

and the starting point is `x0 = [100; -1];`.

First, write an M-file to return the value of `F` (`F` has `n` components).

```
function F = myfun(x,xdata)
F = x(1)*exp(x(2)*xdata);
```

Next, invoke an optimization routine:

```
% Assume you determined xdata and ydata experimentally
xdata = ...
 [0.9 1.5 13.8 19.8 24.1 28.2 35.2 60.3 74.6 81.3];
ydata = ...
 [455.2 428.6 124.1 67.3 43.2 28.1 13.1 -0.4 -1.3 -1.5];
x0 = [100; -1] % Starting guess
[x,resnorm] = lsqcurvefit(@myfun,x0,xdata,ydata)
```

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 27 function evaluations, this example gives the solution

```
x =
```

```
498.8309 -0.1013
resnorm =
9.5049
```

There may be a slight variation in the number of iterations and the value of the returned `x`, depending on the platform and release.

Algorithm

Trust-Region-Reflective Optimization

By default `lsqcurvefit` chooses the trust-region-reflective algorithm. This algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [1] and [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See “Large-Scale Least Squares” on page 4-117, and in particular, “Large Scale Nonlinear Least Squares” on page 4-119.

Levenberg-Marquardt Optimization

If you set the `Algorithm` option to `'levenberg-marquardt'` with `optimset`, `lsqcurvefit` uses the Levenberg-Marquardt method [4], [5], and [6]. See “Levenberg-Marquardt Method” on page 4-121.

Gauss-Newton

The Gauss-Newton method [6] is going to be removed in a future version of MATLAB. Currently, you get a warning when using it. It is not a large-scale method.

Select the Gauss-Newton method [3] with line search by setting the options `LevenbergMarquardt` to `'off'`and `LargeScale` to `'off'` with `optimset`. The Gauss-Newton method can be faster than the Levenberg-Marquardt method when the residual `sum((fun(x,xdata)-ydata).^2)` is small.

The default line search algorithm, i.e., the `LineSearchType` option, is `'quadcubic'`. This 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 in "Gauss-Newton Method" on page 4-122.

Diagnostics

Trust-Region-Reflective Optimization

The trust-region-reflective 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.

Trust-Region-Reflective Optimization

The trust-region-reflective 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 Levenberg-Marquardt algorithm is used instead.

The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective 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.

Trust-Region-Reflective Problem Coverage and Requirements

For Large Problems
• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.

Levenberg-Marquardt Optimization

The Levenberg-Marquardt algorithm does not handle bound constraints.

Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems with both these characteristics cannot be solved by `lsqcurvefit`.

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.

See Also

`@ (function_handle)`, `\` (matrix left division), `lsqlin`, `lsqnonlin`, `lsqnonneg`, `optimset`, `optimtool`, `nlinfit`

For more details about the `lsqcurvefit` algorithms, see “Least Squares (Model Fitting)” on page 4-116. For another example using `lsqcurvefit`, see “Example: Nonlinear Curve Fitting with lsqcurvefit” on page 4-139.

Note The Statistics Toolbox function `nlinfit` has more statistics-oriented outputs that are useful, for example, in finding confidence intervals for the coefficients. It also comes with the `nlintool` GUI for visualizing the fitted function. The `lsqnonlin` function has more outputs related to how well the optimization performed. It can put bounds on the parameters, and it accepts many options to control the optimization algorithm.

---

lsqlin

Purpose

Solve constrained linear least-squares problems

Equation

Solves least-squares curve fitting problems of the form

$$\min_{x} \|C \cdot x - d\|_2^2 \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

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 = lsqlin(problem)
[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 = lsqlin(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-167.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.

Components of `x0` that violate the bounds `lb` ≤ `x` ≤ `ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

If no `x0` is provided, `x0` is set to the zero vector. If any component of this zero vector `x0` violates the bounds, `x0` is set to a point in the interior of the box defined by the bounds.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `lsqlin`. "Options" on page 9-169 provides the `options` values specific to `lsqlin`.

`problem`	`C`	Matrix
	`d`	Vector
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`lb`	Vector of lower bounds
	`ub`	Vector of upper bounds
	`x0`	Initial point for `x`
	`solver`	`'lsqlin'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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	Optimization algorithm used
	cgiterations	Total number of PCG iterations (large-scale algorithm, [ ] for medium-scale)
	firstorderopt	Measure of first-order optimality (large-scale algorithm, [ ] for medium-scale)
	message	Exit message

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 7-7 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; `'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 should compute the Jacobian matrix product `C*Y`, `C'*Y`, or `C'*(C*Y)` without actually forming `C`. Write the function in the form `W = jmfun(Jinfo,Y,flag)` where `Jinfo` contains a matrix used to compute `C*Y` (or `C'*Y`, or `C'*(C*Y)`). `jmfun` must compute one of three different products, depending on the value of `flag` that `lsqlin` passes: • If `flag == 0` then `W = C'*(C*Y)`. • If `flag > 0` then `W = C*Y`. • If `flag < 0` then `W = C'*Y`. In each case, `jmfun` need not form `C` explicitly. `lsqlin` uses `Jinfo` to compute the preconditioner. See "Passing Extra Parameters" on page 2-17 for information on how to supply extra parameters if necessary. See "Example: Jacobian Multiply Function with Linear Least Squares" on page 4-134 for an example.
`MaxPCGIter`	Maximum number of PCG (preconditioned conjugate gradient) iterations (see "Algorithm" on page 9-174).

`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. Setting `PrecondBandWidth` to `'Inf'` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.
`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 \ (matrix left division). 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 4-3 and "Preconditioned Conjugate Gradient Method" on page 4-23.

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 "Large-Scale quadprog Algorithm" on page 4-90.

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 following 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.

Large-Scale Problem Coverage and Requirements

For Large Problems
C should be sparse.

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.

See Also

`\` (matrix left division), `lsqnonneg`, `quadprog`, `optimtool`

For more details about the `lsqlin` algorithms, see "Least Squares (Model Fitting)" on page 4-116. For more examples of least-squares, see "Least Squares (Model Fitting) Examples" on page 4-126.

Purpose

Solve nonlinear least-squares (nonlinear data-fitting) problems

Equation

Solves nonlinear least-squares curve fitting problems of the form

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2\right)$$

Syntax

```
x = lsqnonlin(fun,x0)
x = lsqnonlin(fun,x0,lb,ub)
x = lsqnonlin(fun,x0,lb,ub,options)
x = lsqnonlin(problem)
[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 $\|f(x)\|_2^2$ (the sum of squares), `lsqnonlin` requires the user-defined function to compute the *vector*-valued function

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}$$

Then, in vector terms, you can restate this optimization problem as

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2\right)$$

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)`.)

Note "Passing Extra Parameters" on page 2-17 explains how to pass extra parameters to the vector function *f*, if necessary.

`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 = lsqnonlin(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-179.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.

Components of `x0` that violate the bounds `lb` ≤ `x` ≤ `ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `lsqnonlin`. This section provides function-specific details for `fun`, `options`, and `problem`:

`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 user-defined values for `x` and `F` are matrices, they are converted to a vector using linear indexing.

If the Jacobian can also be computed *and* the Jacobian option is 'on', set by

```
options = optimset('Jacobian','on')
```

the function fun must return, in a second output argument, the Jacobian value J, a matrix, at x. 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, 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). (The Jacobian J is the transpose of the gradient of F.)

options	"Options" on page 9-182 provides the function-specific details for the options values.	
problem	objective	Objective function
	x0	Initial point for x
	lb	Vector of lower bounds
	ub	Vector of upper bounds
	solver	'lsqnonlin'
	options	Options structure created with optimset

Output Arguments

"Function Arguments" on page 7-2 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`	Output function terminated the algorithm.
`-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	
	`firstorderopt`	Measure of first-order optimality (trust-region-reflective algorithm, [ ] for others)
	`iterations`	Number of iterations taken
	`funcCount`	The number of function evaluations
	`cgiterations`	Total number of PCG iterations (trust-region-reflective algorithm, [ ] for others)
	`stepsize`	Final displacement in `x` (Levenberg-Marquardt algorithm)
	`algorithm`	Optimization algorithm used
	`message`	Exit message

Note The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See "Examples" on page 9-187.

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 trust-region-reflective algorithm, and others are only relevant when you are using the Levenberg-Marquardt algorithm. See "Optimization Options" on page 7-7 for detailed information.

The `Algorithm` option specifies a preference for which algorithm to use. It is only a preference, because certain conditions must be met to use the trust-region-reflective or Levenberg-Marquardt algorithm. For the

trust-region-reflective 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 trust-region-reflective algorithm handles bound constraints:

Both algorithms handle both large-scale and medium-scale problems effectively.

Algorithm Options

These options are used by both algorithms:

`Algorithm`	Use trust-region-reflective algorithm if possible when set to `'trust-region-reflective'`. Use Levenberg-Marquardt algorithm when set to `'levenberg-marquardt'`.
`DerivativeCheck`	Compare user-supplied derivatives (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`Display`	Level of display. `'off'` displays no output, and `'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 one or more user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx` plots the current point; `@optimplotfunccount` plots the function count; `@optimplotfval` plots the function value.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

Trust-Region-Reflective Algorithm Only

These options are used only by the trust-region-reflective 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)
```

where `Jinfo` contains the matrix 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, W = J'*(J*Y).
- If flag > 0, W = J*Y.
- If flag < 0, W = J'*Y.

In each case, J is not formed explicitly. fsolve uses Jinfo to compute the preconditioner. See "Passing Extra Parameters" on page 2-17 for information on how to supply values for any additional parameters jmfun needs.

Note 'Jacobian' must be set to 'on' for Jinfo to be passed from fun to jmfun.

See "Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64 and "Example: Jacobian Multiply Function with Linear Least Squares" on page 4-134 for similar examples.

`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 9-162).
`PrecondBandWidth`	The default `PrecondBandWidth` is `'Inf'`, which means a direct factorization (Cholesky) is used rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution. Set `PrecondBandWidth` to 0 for diagonal preconditioning (upper bandwidth of 0). For some problems, an intermediate bandwidth reduces the number of PCG iterations.
`TolPCG`	Termination tolerance on the PCG iteration.

Levenberg-Marquardt Algorithm Only

This option is used only by the Levenberg-Marquardt algorithm:

ScaleProblem	'Jacobian' can sometimes improve the convergence of a poorly-scaled problem; the default is 'none'.

Gauss-Newton Algorithm Only

These options are used only by the Gauss-Newton algorithm:

LargeScale and LevenbergMarquardt	Specify LargeScale as 'off' and LevenbergMarquardt as 'off' to choose the Gauss-Newton algorithm. These options are being obsoleted, and are not used for the other algorithms.
LineSearchType	The choices are 'cubicpoly' or the default 'quadcubic'.

Examples

Find x that minimizes

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

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

Because lsqnonlin assumes that the sum of squares is *not* explicitly formed in the user-defined 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

Trust-Region-Reflective Optimization

By default, `lsqnonlin` chooses the trust-region-reflective algorithm. This algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [1] and [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See "Large-Scale Least Squares" on page 4-117, and in particular "Large Scale Nonlinear Least Squares" on page 4-119.

Levenberg-Marquardt Optimization

If you set the `Algorithm` option to `'levenberg-marquardt'` using `optimset`, `lsqnonlin` uses the Levenberg-Marquardt method [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 4-121.

Gauss-Newton

The Gauss-Newton method [6] is going to be removed in a future version of MATLAB. Currently, you get a warning when using it. It is not a large-scale method.

Select the Gauss-Newton method [3] with line search by setting the options `LevenbergMarquardt` to `'off'` and `LargeScale` to `'off'` with `optimset`. The Gauss-Newton method can be faster than the

Levenberg-Marquardt method when the residual $\|F(x)\|_2^2$ is small. See “Gauss-Newton Method” on page 4-122.

The default line search algorithm, i.e., the `LineSearchType` option, is `'quadcubic'`. This 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.

Diagnostics

Trust-Region-Reflective Optimization

The trust-region-reflective 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 (the user-defined function must only return real values). When `x` has complex variables, the variables must be split into real and imaginary parts.

Trust-Region-Reflective Optimization

The trust-region-reflective algorithm for `lsqnonlin` 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 Levenberg-Marquardt algorithm is used instead.

The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective method forms $J^T J$ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product $J^T J$, can lead to a costly solution process for large problems.

If components of x have no upper (or lower) bounds, `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.

Trust-Region-Reflective Problem Coverage and Requirements

For Large Problems
• Provide sparsity structure of the Jacobian or compute the Jacobian in `fun`. • The Jacobian should be sparse.

Levenberg-Marquardt Optimization

The Levenberg-Marquardt algorithm does not handle bound constraints.

Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems with both these characteristics cannot be solved by `lsqnonlin`.

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.

See Also

`@ (function_handle)`, `lsqcurvefit`, `lsqlin`, `optimset`, `optimtool`

For more details about the `lsqnonlin` algorithms, see “Least Squares (Model Fitting)” on page 4-116. For more examples of least squares, see “Least Squares (Model Fitting) Examples” on page 4-126.

lsqnonneg

Purpose Solve nonnegative least-squares constraint problem

Equation Solves nonnegative least-squares curve fitting problems of the form

$$\min_{x} \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,x0)
x = lsqnonneg(C,d,x0,options)
x = lsqnonneg(problem)
[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 = lsqnonneg(problem)` finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 9-193.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in “Exporting to the MATLAB Workspace” on page 3-14.

`[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 7-2 contains general descriptions of arguments passed into `lsqnonneg`. This section provides function-specific details for `options` and `problem`:

`options`	Use `optimset` to set or change the values of these fields in the options structure, `options`. See "Optimization Options" on page 7-7 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`.
`problem`	`C`	Matrix
	`d`	Vector
	`x0`	Initial point for `x`
	`solver`	`'lsqnonneg'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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` — Optimization algorithm used `message` — Exit message

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.

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least-Squares Problems*, Prentice-Hall, Chapter 23, p. 161, 1974.

See Also

`\` (matrix left division), `lsqlin`, `optimset`, `optimtool`

optimget

Purpose

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 7-7 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:

`Algorithm`	`'active-set'` | `'interior-point'` | {`'trust-region-reflective'`}
`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`	`function_handle` | `{[]}`. Specify one or more (using a cell array of function handles) user-defined functions that an optimization function calls at each iteration. See "Output Function" on page 7-17.

PlotFcns	function_handle | {[]}. Specify one or more (using a cell array of function handles) pre- or user-defined plot functions that an optimization function calls at each iteration. See "Plot Functions" on page 7-26.
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 The default value depends on the solver.
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}
FinDiffType	'central' | {'forward'}
GoalsExactAchieve	Positive scalar integer | {0}
GradConstr	'on' | {'off'}
HessUpdate	{'bfgs'} | 'dfp' | 'steepdesc'
LevenbergMarquardt	Choose Gauss-Newton algorithm for lsqcurvefit or lsqnonlin by setting to 'off', and also setting LargeScale to '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	Choose Gauss-Newton algorithm for fsolve by setting to 'gn', and also setting LargeScale to 'off'.
RelLineSrchBnd	Real nonnegative scalar | {[]}
RelLineSrchBndDuration	Positive integer | {1}

Simplex	When you set 'Simplex' to 'on' and 'LargeScale' to 'off', linprog uses the simplex algorithm to solve a constrained linear programming problem.
TolConSQP	Positive scalar | {1e-6}
TolRLPFun	Positive scalar | {1e-6}
TolXInteger	Positive scalar | {1e-8}
UseParallel	'always' | {'never'}

Optimization options used by interior-point algorithm only:

AlwaysHonorConstraints	'none' | {'bounds'}
FinDiffType	'central' | {'forward'}
HessFcn	Function handle to a user-supplied Hessian
Hessian	'fin-diff-grads' | 'lbfgs' | {'lbfgs',Positive Integer} | 'user-supplied' | {bfgs}
HessMult	Handle to a user-supplied function that gives a Hessian-times-vector product
InitBarrierParam	Positive real scalar | {0.1}
InitTrustRegionRadius	Positive real scalar | { $\sqrt{n}$ }, where n is the number of variables.
MaxProjCGIter	Positive integer | {2*(n - neq}, where n is the number of variables, and neq is the number of equalities
ObjectiveLimit	Scalar | -1e20
ScaleProblem	'none' | {'obj-and-constr'
SubproblemAlgorithm	'cg' | {'ldl-factorization'}

TolProjCG	Positive scalar | {1e-2}
TolProjCGAbs	Positive scalar | {1e-10}

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, optimtool

optimtool

Purpose

Tool to select solver, optimization options, and run problems

Syntax

```
optimtool
optimtool(optstruct)
optimtool('solver')
```

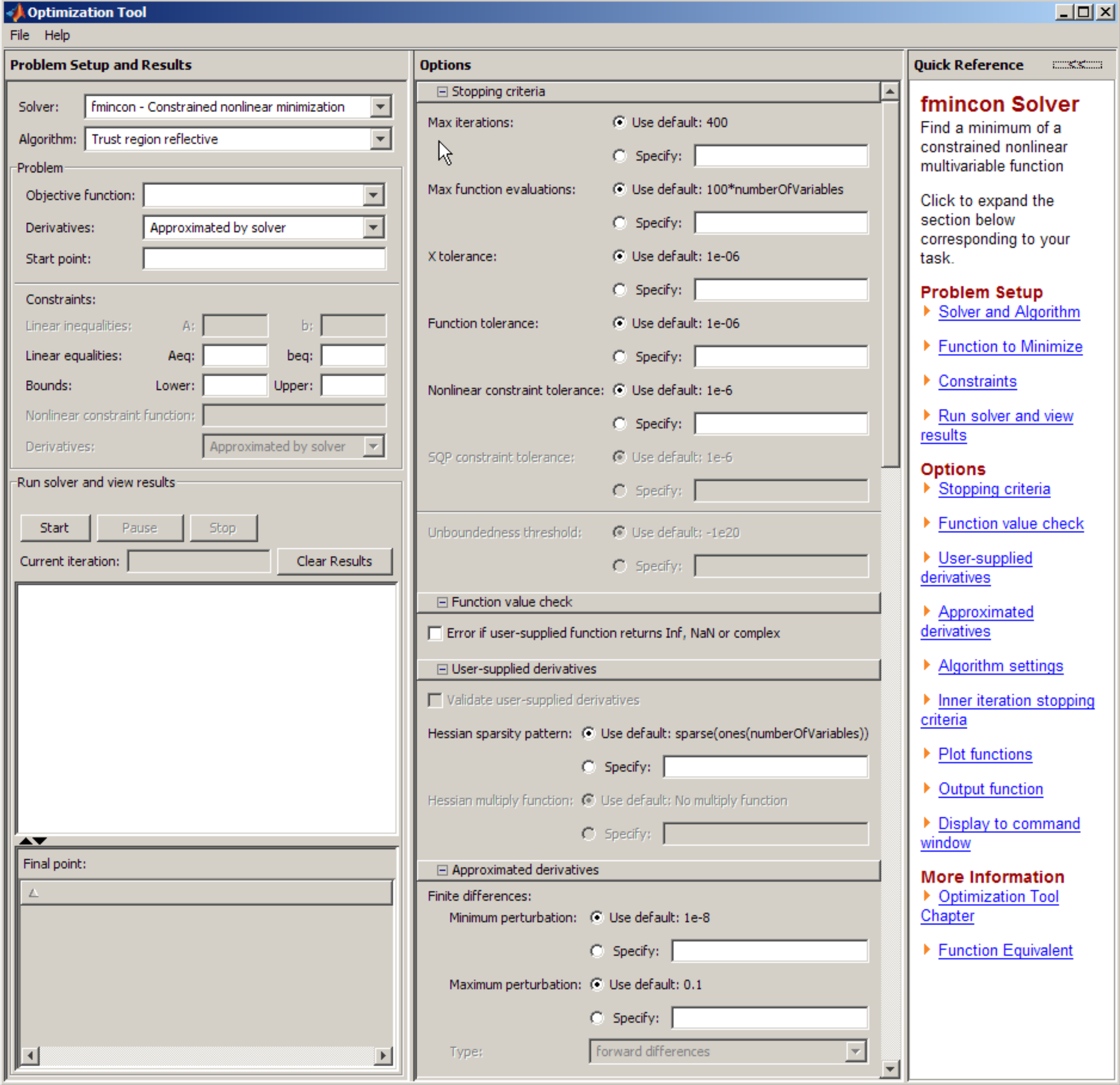
Description

`optimtool` opens the Optimization tool, a graphical user interface (GUI) for selecting a solver, the optimization options, and running problems. See Chapter 3, "Optimization Tool" for a complete description of the tool.

`optimtool` can be used to run any Optimization Toolbox solver, and any Genetic Algorithm and Direct Search Toolbox solver. Results can be exported to an M-file or to the MATLAB workspace as a structure.

`optimtool(optstruct)` starts the Optimization Tool with `optstruct`. `optstruct` can either be an optimization options structure or optimization problem structure. An options structure can be created using the `optimset` function or by using the export option from `optimtool`. A problem structure can be created or modified in `optimtool` by exporting the problem information to the MATLAB workspace.

`optimtool('solver')` starts the Optimization Tool with the specified solver, identified as a string, and the corresponding default options and problem fields. All Optimization Toolbox solvers are valid inputs to the `optimtool` function.


Optimization Tool
File Help
Problem Setup and Results
Solver: fmincon - Constrained nonlinear minimization
Algorithm: Trust region reflective
Problem
Objective function:
Derivatives: Approximated by solver
Start point:
Constraints:
Linear inequalities: A: b:
Linear equalities: Aeq: beq:
Bounds: Lower: Upper:
Nonlinear constraint function:
Derivatives: Approximated by solver
Run solver and view results
Start Pause Stop
Current iteration: Clear Results
Final point:
Options
Stopping criteria
Max iterations: Use default: 400
Specify:
Max function evaluations: Use default: 100*numberOfVariables
Specify:
X tolerance: Use default: 1e-06
Specify:
Function tolerance: Use default: 1e-06
Specify:
Nonlinear constraint tolerance: Use default: 1e-6
Specify:
SQP constraint tolerance: Use default: 1e-6
Specify:
Unboundedness threshold: Use default: -1e20
Specify:
Function value check
Error if user-supplied function returns Inf, NaN or complex
User-supplied derivatives
Validate user-supplied derivatives
Hessian sparsity pattern: Use default: sparse(ones(numberOfVariables))
Specify:
Hessian multiply function: Use default: No multiply function
Specify:
Approximated derivatives
Finite differences:
Minimum perturbation: Use default: 1e-8
Specify:
Maximum perturbation: Use default: 0.1
Specify:
Type: forward differences
Quick Reference
fmincon Solver
Find a minimum of a constrained nonlinear multivariable function
Click to expand the section below corresponding to your task.
Problem Setup
Solver and Algorithm
Function to Minimize
Constraints
Run solver and view results
Options
Stopping criteria
Function value check
User-supplied derivatives
Approximated derivatives
Algorithm settings
Inner iteration stopping criteria
Plot functions
Output function
Display to command window
More Information
Optimization Tool Chapter
Function Equivalent

See Also

`optimset`

Purpose

Solve quadratic programming problems

Equation

Finds a minimum for a problem specified by

$$\min_x \frac{1}{2} x^T H x + f^T x \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

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 = quadprog(problem)
[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`. If no inequalities exist, set `A = []` and `b = []`.

`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`. If no equalities exist, set `Aeq = []` and `beq = []`.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0)` sets the starting point to `x0`. If no bounds exist, set `lb = []` and `ub = []`.

`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. If you do not wish to give an initial point, set `x0 = []`.

`x = quadprog(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 9-209.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in "Exporting to the MATLAB Workspace" on page 3-14.

`[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`.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

Components of `x0` that violate the bounds `lb` ≤ `x` ≤ `ub` are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

If no `x0` is provided, all components of `x0` are set to a point in the interior of the box defined by the bounds.

Input Arguments

"Function Arguments" on page 7-2 contains general descriptions of arguments passed into `quadprog`. "Options" on page 9-211 provides function-specific details for the `options` values.

`problem`	`H`	Symmetric matrix
	`f`	Vector
	`Aineq`	Matrix for linear inequality constraints
	`bineq`	Vector for linear inequality constraints
	`Aeq`	Matrix for linear equality constraints
	`beq`	Vector for linear equality constraints
	`lb`	Vector of lower bounds
	`ub`	Vector of upper bounds
	`x0`	Initial point for `x`
	`solver`	`'quadprog'`
	`options`	Options structure created with `optimset`

Output Arguments

"Function Arguments" on page 7-2 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`	Optimization algorithm used

`cgiterations`	Total number of PCG iterations (large-scale algorithm only)
`firstorderopt`	Measure of first-order optimality (large-scale algorithm only)
`message`	Exit message

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 7-7 for detailed information.

The option to set an algorithm preference is as follows.

`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, and '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)` where Hinfo and possibly some additional parameters contain the matrices used to compute H*Y. See "Example: Quadratic Minimization with a Dense but Structured Hessian" on page 4-102 for an example that uses this option.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations. See "Algorithm" on page 9-216 for more information.

`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. Setting `PrecondBandWidth` to `'Inf'` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.
`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

$$\begin{aligned} x_1 + x_2 &\le 2 \\ -x_1 + 2x_2 &\le 2 \\ 2x_1 + x_2 &\le 3 \\ 0 \le x_1,\ 0 &\le x_2. \end{aligned}$$

First, note that this function can be written in matrix notation as

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

where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix},\ f = \begin{bmatrix} -2 \\ -6 \end{bmatrix},\ x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

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: []
          message: 'Optimization terminated.'
lambda =
      lower: [2x1 double]
```

```
         upper: [2x1 double]
         eqlin: [0x1 double]
       ineqlin: [3x1 double]

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

By default, `quadprog` uses the large-scale algorithm if you 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 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 4-3 and “Preconditioned Conjugate Gradient Method” on page 4-23.

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 “Medium-Scale quadprog Algorithm” on page 4-95.

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 this 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 problem 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.

Large-Scale Problem Coverage and Requirements

For Large Problems
• *H* should be sparse. • *Aeq* should be sparse.

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.

See Also

`linprog`, `lsqlin`, `optimtool`

For more details about the `quadprog` algorithms, see "Quadratic Programming" on page 4-90. For more examples of quadratic programming, see "Quadratic Programming Examples" on page 4-100.

Examples

Use this list to find examples in the documentation.

Constrained Nonlinear Examples

"Example: Nonlinear Constrained Minimization" on page 1-4
"An Example Using All Types of Constraints" on page 2-15
"Optimization Tool with the fmincon Solver" on page 3-18
"Example: Nonlinear Inequality Constraints" on page 4-44
"Example: Bound Constraints" on page 4-46
"Example: Constraints With Gradients" on page 4-47
"Example: Constrained Minimization Using fmincon's Interior-Point Algorithm With Analytic Hessian" on page 4-50
"Example: Equality and Inequality Constraints" on page 4-57
"Example: Nonlinear Minimization with Bound Constraints and Banded Preconditioner" on page 4-58
"Example: Nonlinear Minimization with Equality Constraints" on page 4-62
"Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints" on page 4-64
"Example: One-Dimensional Semi-Infinite Constraints" on page 4-68
"Example: Two-Dimensional Semi-Infinite Constraint" on page 4-70
"Example Using ktrlink" on page 6-4

Least Squares Examples

"Optimization Tool with the lsqlin Solver" on page 3-22
"Example: Using lsqnonlin With a Simulink Model" on page 4-126
"Example: Nonlinear Least-Squares with Full Jacobian Sparsity Pattern" on page 4-131
"Example: Linear Least-Squares with Bound Constraints" on page 4-133
"Example: Jacobian Multiply Function with Linear Least Squares" on page 4-134
"Example: Nonlinear Curve Fitting with lsqcurvefit" on page 4-139

Unconstrained Nonlinear Examples

"Example: fminunc Unconstrained Minimization" on page 4-14

"Example: Nonlinear Minimization with Gradient and Hessian" on page 4-16
"Example: Nonlinear Minimization with Gradient and Hessian Sparsity Pattern" on page 4-17

Linear Programming Examples

"Example: Linear Programming with Equalities and Inequalities" on page 4-86
"Example: Linear Programming with Dense Columns in the Equalities" on page 4-87

Quadratic Programming Examples

"Example: Quadratic Minimization with Bound Constraints" on page 4-100
"Example: Quadratic Minimization with a Dense but Structured Hessian" on page 4-102

Binary Integer Programming Examples

"Example: Investments with Constraints" on page 4-111

Multiobjective Examples

"Example: Using fminimax with a Simulink Model" on page 4-147
"Example: Signal Processing Using fgoalattain" on page 4-150

Equation Solving Examples

"Example: Nonlinear Equations with Analytic Jacobian" on page 4-162

“Example: Nonlinear Equations with Finite-Difference Jacobian” on page 4-165
“Example: Nonlinear Equations with Jacobian” on page 4-166
“Example: Nonlinear Equations with Jacobian Sparsity Pattern” on page 4-169

Index

A

active constraints
    `linprog` example 9-144
    `lsqlin` example 9-173
    `quadprog` example 9-215
active set method
    `fmincon` medium-scale algorithm 9-55
    `linprog` medium-scale algorithm 9-145
    `lsqlin` medium-scale algorithm 9-174
    `quadprog` medium-scale algorithm 9-216
    sequential quadratic programming (SQP) in fmincon 4-31
    sequential quadratic programming (SQP) in linprog 4-78
    sequential quadratic programming (SQP) in quadprog 4-96
attainment factor 9-24
Avoiding Global Variables 2-17
axis crossing.. *See* zero of a function

B

banana function. *See* Rosenbrock's function
Barrier function 4-35
basin of attraction 2-57
BFGS formula 4-8
    `fmincon` medium-scale algorithm 9-55
    `fminunc` medium-scale algorithm 9-91
`bintprog` 9-2
bisection search 9-126
bound constraints, large-scale 4-24
box constraints.. *See* bound constraints

C

centering parameter 4-76
CG. *See* conjugate gradients
color 9-9
complementarity conditions 4-75
complex variables
    `lsqcurvefit` 9-163
    `lsqnonlin` 9-189
computing, parallel 5-1
conjugate gradients
    in fmincon 4-22
    in fmincon interior-point 4-36
    in fminunc 4-5
    in fsolve 4-158
    in least squares 4-118
    in quadprog 4-91
constrained minimization 9-129
    fmincon 9-34
    large-scale example
        with bound constraints and banded preconditioner 4-58
        with equality constraints 4-62
constraints
    linear 4-23
        `fmincon` 9-55
        `fminimax` 9-70
continuous derivative
    gradient methods 4-6
convex problem 4-26
curve-fitting
    categories 2-23
    functions that apply 8-3
    lsqcurvefit 9-150

D

data-fitting 9-150
    categories 2-23
    functions that apply 8-3
dense columns, constraint matrix 4-76
DFP formula 9-91

direction of negative curvature
  fmincon 4-22
  fminunc 4-5
  fsolve 4-158
  least squares 4-119
  quadprog 4-92
discontinuities 2-55
discontinuous problems
  `fminsearch` 9-79
  `fminunc` 9-92
discrete variables 2-56
distributed computing 5-1
dual problem 4-75
duality gap 4-75

E

equality constraints
  dense columns 4-87
  medium-scale example 4-57
equality constraints inconsistent warning,
  `quadprog` 9-217
equality constraints, linear
  large-scale 4-23
equation solving 4-160
  categories 2-23
  functions that apply 8-2
error, `Out of memory` 4-131

F

F-count 2-27
feasibility conditions 4-75
feasible point, finding
  fmincon 4-34
  linprog 4-81
  quadprog 4-98
`fgoalattain` 9-10
  example 4-151
first-order optimality measure 2-28
fixed variables 4-77
fixed-step ODE solver 4-131
flag. *See* exitflag on individual function pages
`fminbnd` 9-28
`fmincon` 9-34
  large-scale example
    with bound constraints and banded
      preconditioner 4-58
    with equality constraints 4-62
  medium-scale example 4-44
`fminimax` 9-59
  example 4-147
`fminsearch` 9-73
`fminunc` 9-80
  large-scale example 4-16
  medium-scale example 4-14
  warning messages 2-54
`fseminf` 9-94
`fsolve` 9-106
  analytic Jacobian 4-162
  finite difference Jacobian 4-165
  Jacobian 4-166
function arguments 7-2
function discontinuities 2-55
functions
  grouped by category 8-1
`fzero` 9-121
`fzmult` 9-127

G

`gangstr` 9-128
Gauss-Newton method (large-scale)
  nonlinear least-squares 4-120
Gauss-Newton method (medium-scale)
  implementation, nonlinear least
    squares 4-122
  least-squares optimization 4-122
  solving nonlinear equations 4-160
global and local minima 2-57

global optimum 5-9
goal attainment 4-142
  example 4-151
  `fgoalattain` 9-10
`goaldemo` 9-23
golden section search 9-33
Gradient
  writing 2-4
gradient examples 4-47
gradient methods
  continuous first derivative 4-6
  quasi-Newton 4-8
  unconstrained optimization 4-6

H

Hessian 2-4
  writing 2-4
`Hessian modified` message 4-30
`Hessian modified twice` message 4-30
Hessian sparsity structure 4-18
Hessian update 4-11
  stage of SQP implementation 4-29
Hessian updating methods 4-8

I

inconsistent constraints 9-148
indefinite problems 9-218
`infeasible` message 4-31
infeasible optimization problems 2-55
infeasible problems 9-56
infeasible solution warning
  `linprog` 9-147
  `quadprog` 9-217
input arguments 7-2
integer variables 2-56
interior-point algorithm 4-50
interior-point linear programming 4-74
introduction to optimization 4-2

J

Jacobian 2-6
  analytic 4-162
  finite difference 4-165
  nonlinear equations 4-166
Jacobian Multiply Function 4-134
Jacobian sparsity pattern 4-169

K

Karush-Kuhn-Tucker conditions 2-29
  fmincon 4-26
KKT conditions 2-29
KNITRO® 6-2
`ktrlink` 9-129

L

Lagrange multiplier
  fmincon interior point Hessian example 4-52
Lagrange multiplier structures 2-32
Lagrange multipliers
  large-scale linear programming 4-78
Lagrangian
  definition 2-29
  in fmincon algorithms 9-41
  ktrlink 9-134
Large-scale algorithm 2-45
least squares 4-124
  categories 2-23
  functions that apply 8-3
Levenberg-Marquardt method
  equation solving 4-160
  least squares 4-121
line search
  `fminunc` medium-scale default 9-91
  unconstrained optimization 4-10
linear constraints 4-23
  `fmincon` 9-55
  `fminimax` 9-70

linear equations solve 9-117
linear least squares
  constrained 9-166
  large-scale algorithm 4-120
  large-scale example 4-133
  nonnegative 9-192
  unconstrained 9-174
linear programming 9-139
  in fmincon 4-34
  in quadprog 4-98
  large-scale algorithm 4-74
  large-scale example
    with dense columns in equalities 4-87
    with equalities and inequalities 4-86
  problem 4-2
`linprog` 9-139
  large-scale example
    with dense columns in equalities 4-87
    with equalities and inequalities 4-86
LIPSOL 4-74
local and global minima 2-57
lower bounds 4-46
`lsqcurvefit` 9-150
`lsqlin` 9-166
  large-scale example 4-133
`lsqnonlin` 9-177
  convergence 2-56
  large-scale example 4-169
  medium-scale example 4-128
`lsqnonneg` 9-192

M

Maximization 2-9
Medium-scale algorithm 2-45
Mehrotra's predictor-corrector algorithm 4-74
merit function
  definition 4-36
  fmincon 4-35
minimax examples 4-147
minimax problem, solving 9-59
minimization
  categories 2-21
  functions that apply 8-2
minimum
  global 2-57
  local 2-57
Multiobjective
  categories 2-22
multiobjective optimization
  `fgoalattain` 9-10

N

negative curvature direction
  in PCG algorithm 4-6 4-23 4-93 4-159
  in trust-region methods 4-5 4-22 4-92 4-119 4-158
negative definite problems 9-218
Nelder and Mead 4-6
Newton direction
  approximate 4-5 4-22 4-92 4-118 4-158
Newton's method
  systems of nonlinear equations 4-155
  unconstrained optimization 4-6
`no update` message 4-31
nonconvex problems 9-218
nonlinear data-fitting
  `fminsearch` 9-79
  `fminunc` 9-91
  `lsqnonlin` 9-177
nonlinear equations
  analytic Jacobian example 4-162
  example with Jacobian 4-166
  finite difference Jacobian example 4-165
  Newton's method 4-155
nonlinear equations (large-scale)
  solving 9-106

nonlinear equations (medium-scale)
  Gauss-Newton method 4-160
  solving 9-106
  trust-region dogleg method 4-155
nonlinear least squares 4-121 to 4-122
  `fminsearch` 9-79
  `fminunc` 9-91
  large-scale algorithm 4-119
  large-scale example 4-169
  `lsqcurvefit` 9-150
  `lsqnonlin` 9-177
nonlinear programming 4-2

O

objective function
  return values 2-56
optimality conditions linear programming 4-75
Optimality measure
  first-order 2-28
optimality measure, first-order 2-29
`optimget` 9-197
optimization
  functions by category 8-1
  handling infeasibility 2-55
  helpful hints 2-54
  introduction 4-2
  objective function return values 2-56
  troubleshooting 2-54
  unconstrained 4-6
optimization parameters structure 2-41
  `optimget` 9-197
  `optimset` 9-198
Optimization Tool 3-1
  `functions that apply` 8-3
  opening 3-2
  `optimtool` 9-204
  options 3-10
  pausing and stopping 3-7
  running a problem 3-6
  steps 3-5
`optimset` 9-198
`optimtool` 9-204
optimum, global 5-9
options parameters
  descriptions 7-7
  possible values 9-199
  utility functions 8-4
`Out of memory` error 4-131
output arguments 7-2
output display 2-47
output function 2-33
Output structure 2-33

P

parallel computing 5-1
Parameters, Additional 2-17
PCG. *See* preconditioned conjugate gradients
preconditioned conjugate gradients
  algorithm 4-6 4-23 4-93 4-159
  in fmincon 4-22
  in fminunc 4-5
  in fsolve 4-158
  in least squares 4-118
  in quadprog 4-91
preconditioner 4-168
  banded 4-58
  in PCG method 4-6 4-23 4-93 4-159
predictor-corrector algorithm 4-76
preprocessing 4-77
  linear programming 4-74
primal problem 4-75

primal-dual algorithm 4-75
primal-dual interior-point 4-74
projection method
  quadprog medium-scale algorithm 9-216
  sequential quadratic programming (SQP) in fmincon 4-31
  sequential quadratic programming (SQP) in linprog 4-78
  sequential quadratic programming (SQP) in quadprog 4-96

Q

quadprog 9-207
  large-scale example 4-100
quadratic programming 4-2
  fmincon 9-55
  large-scale algorithm 4-90
  large-scale example 4-100
  quadprog 9-207
quasi-Newton methods 4-8
  fminunc medium-scale algorithm 9-91
  unconstrained optimization 4-8

R

reflective line search 4-92
reflective steps
  fmincon 4-24
  fmincon definition 4-25
  quadprog 4-94
  quadprog definition 4-95
residual 4-123
revised simplex algorithm 4-82
Rosenbrock's function
  anonymous function implementation 2-9
  fminsearch 9-76
  fminunc 4-6
  Gauss-Newton method 4-124
  getting started example 1-4
  Levenberg-Marquardt example 4-122
  multidimensional 4-165
  multidimensional with Jacobian 4-162
  QP example 4-28
  Quasi-Newton method 4-9
  with gradient and Hessian 2-5

S

sampling interval 9-103
secular equation
  fmincon 4-21
  fminunc 4-4
  fsolve 4-158
  least squares 4-118
  quadprog 4-91
semi-infinite constraints 9-94
Sherman-Morrison formula 4-76
signal processing example 4-150
simple bounds 4-46
simplex search 9-78
  unconstrained optimization 4-6
Simulink®, multiobjective example 4-126
singleton rows 4-78
slack 4-36
sparsity pattern, Jacobian 4-169
sparsity structure, Hessian 4-18
SQP method 4-27
  fmincon 9-55
  fmincon implementation 4-31
  linprog implementation 4-78
steepest descent 9-92
stopping criteria 2-31

stopping criteria, large-scale linear
  programming 4-77
structural rank 4-77
subspace
  determination of in fmincon 4-22
  determination of in fminunc 4-5
  determination of in fsolve 4-158
  determination of in least squares 4-118
  determination of in quadprog 4-91
systems of nonlinear equations
  solving 9-106

T

trust region 4-3
trust-region dogleg method (medium-scale)
  systems of nonlinear equations 4-155

U

unbounded solutions warning
  linprog 9-148
  quadprog 9-217
unconstrained minimization
  fminsearch 9-73
  fminunc 9-80
  large-scale example 4-16
  medium-scale example 4-14
  one dimensional 9-28
unconstrained optimization 4-6
upper bounds 4-46

V

variable-step ODE solver 4-131
Variables, Additional 2-17

W

warning
  equality constraints inconsistent,
    quadprog 9-217
  infeasible solution, linprog 9-147
  infeasible solution, quadprog 9-217
  stuck at minimum, fsolve 9-118
  unbounded solutions, linprog 9-148
  unbounded solutions, quadprog 9-217
warnings displayed 2-55

Z

zero curvature direction 4-6 4-23 4-93 4-159
zero finding 9-106
zero of a function, finding 9-121