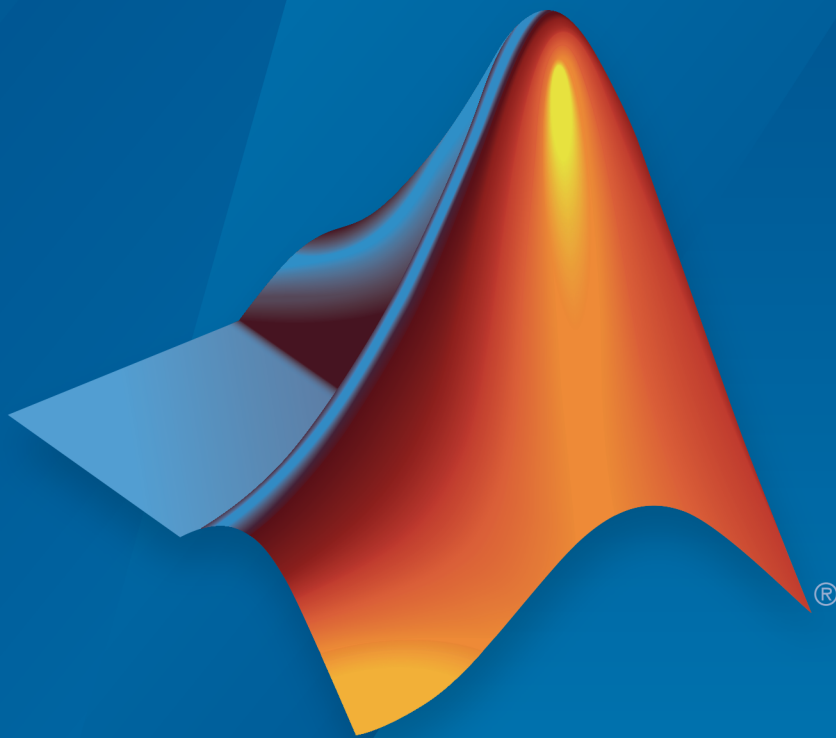# Optimization Toolbox™

User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Optimization Toolbox™ User's Guide

© COPYRIGHT 1990–2015 by The MathWorks, Inc.

Trademarks

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

Patents

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

Contents

Examining Results

3

Steps to Take After Running a Solver

4

Optimization App

5

Nonlinear algorithms and examples

6

7

Multiobjective Algorithms and Examples

8

Linear Programming and Mixed-Integer Linear Programming

Quadratic Programming

9

<div align="right">

Least Squares

</div>

10

Parallel Computing for Optimization

12

Argument and Options Reference

13

Acknowledgments

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

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

Dr. Coleman is Professor of Combinatorics and Optimization at the University of Waterloo.

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

Dr. Zhang is Professor of Computational and Applied Mathematics at Rice University.

Getting Started

- "Optimization Toolbox Product Description" on page 1-2
- "Solve a Constrained Nonlinear Problem" on page 1-3
- "Set Up a Linear Program" on page 1-12

Optimization Toolbox Product Description

Solve linear, quadratic, integer, and nonlinear optimization problems

Optimization Toolbox provides functions for finding parameters that minimize or maximize objectives while satisfying constraints. The toolbox includes solvers for linear programming, mixed-integer linear programming, quadratic programming, nonlinear optimization, and nonlinear least squares. You can use these solvers to find optimal solutions to continuous and discrete problems, perform tradeoff analyses, and incorporate optimization methods into algorithms and applications.

Key Features

- Nonlinear and multiobjective optimization
- Solvers for nonlinear least squares, data fitting, and nonlinear equations
- Quadratic and linear programming
- Mixed-integer linear programming
- Optimization app for defining and solving optimization problems and monitoring solution progress
- Acceleration of constrained nonlinear solvers with Parallel Computing Toolbox™

Solve a Constrained Nonlinear Problem

In this section...
"Problem Formulation: Rosenbrock's Function" on page 1-3
"Defining the Problem in Toolbox Syntax" on page 1-4
"Running the Optimization" on page 1-5
"Interpreting the Result" on page 1-10

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 $x_1^2 + x_2^2 \leq 1$. 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 views 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.

Minimum at (.7864, .6177)
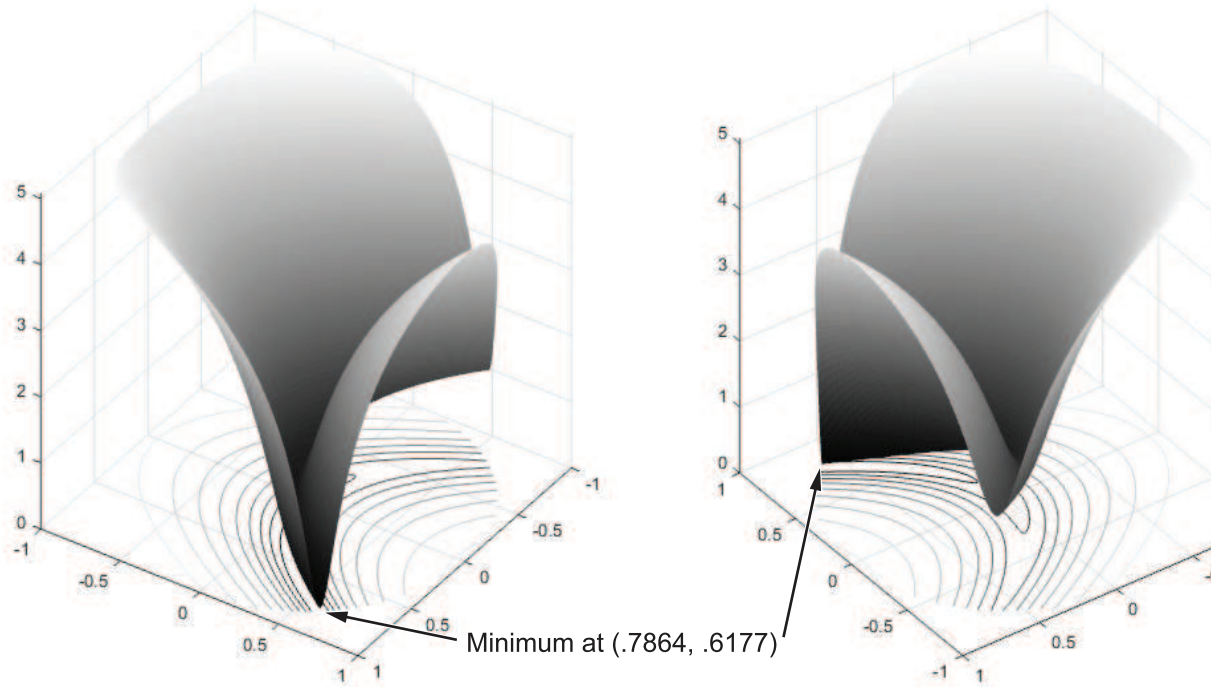
Rosenbrock's function, log-scaled: two views.

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

All Optimization Toolbox optimization functions minimize an objective function. To maximize a function f, apply an optimization routine to minimize $-f$. For more details about maximizing, see "Maximizing an Objective" on page 2-28.

Defining the Problem in Toolbox Syntax

To use Optimization Toolbox software, you need to

1   Define your objective function in the MATLAB® language, as a function file or anonymous function. This example will use a function file.

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

Function File for Objective Function

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

1   At the command line enter:

```
edit rosenbrock
```
The MATLAB Editor opens.

2   In the editor enter:

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

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

File for Constraint Function

Constraint functions must be formulated so that they are in the form $c(x) \leq 0$ or $ceq(x) = 0$. The constraint $x_1^2 + x_2^2 \leq 1$ needs to be reformulated as $x_1^2 + x_2^2 - 1 \leq 0$ 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 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 app" on page 1-6
- Using command line functions; see "Minimizing at the Command Line" on page 1-10.

Optimization app

Note: The Optimization app warns that it will be removed in a future release.

1 Start the Optimization app by typing `optimtool` at the command line.
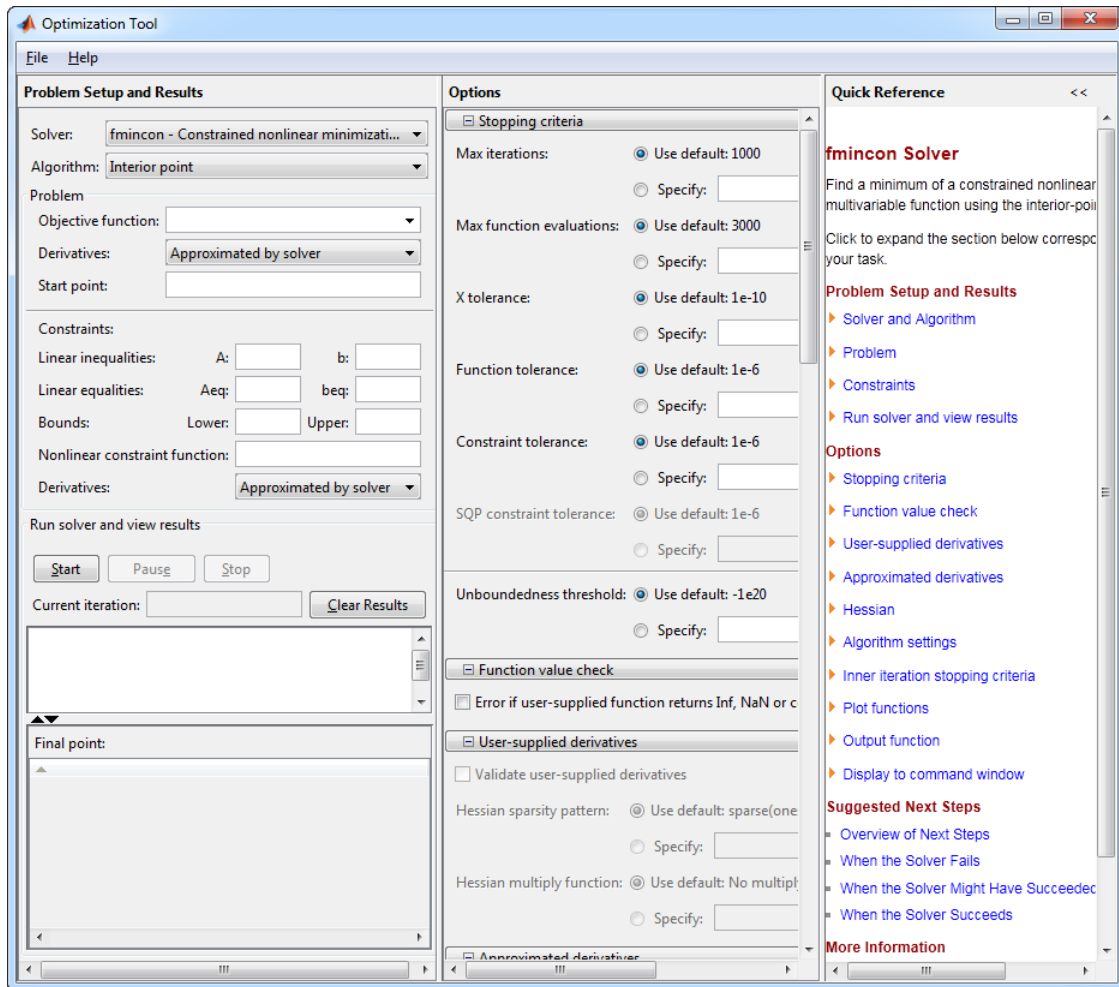
For more information about this tool, see "Optimization App" on page 5-2.

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 2-5.

3 In the **Algorithm** pop-up menu choose `Interior point`, which is the default.

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

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

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

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

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.
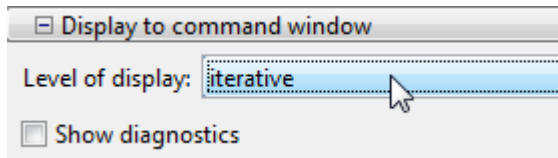
8 Click **Start** under **Run solver and view results**.
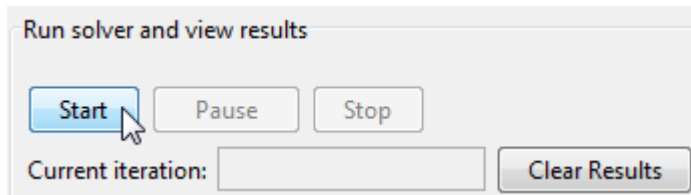
The following message appears in the box below the **Start** button:

```
Optimization running.
Objective function value: 0.045674824758137236
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.
```
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 satisfied to the requisite accuracy.

"Exit Flags and Exit Messages" on page 3-3 discusses exit messages such as these.

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

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 `interior-point` algorithm:

```
options = optimoptions(@fmincon,...
    'Display','iter','Algorithm','interior-point');
```

2 Run the `fmincon` solver with the `options` structure, 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:

```
Local minimum found that satisfies the constraints.

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

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 satisfied to the requisite accuracy. Several phrases in the message contain links that give you more information about the terms used in the message. For more details about these links, see "Enhanced Exit Messages" on page 3-5.

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 app or the command line. MATLAB reports the minimization as follows:

| | | | | First-order | Norm of |
Iter	F-count	f(x)	Feasibility	optimality	step
0	3	1.000000e+00	0.000e+00	2.000e+00	
1	13	7.753537e-01	0.000e+00	6.250e+00	1.768e-01
2	18	6.519648e-01	0.000e+00	9.048e+00	1.679e-01
3	21	5.543209e-01	0.000e+00	8.033e+00	1.203e-01
4	24	2.985207e-01	0.000e+00	1.790e+00	9.328e-02
5	27	2.653799e-01	0.000e+00	2.788e+00	5.723e-02
6	30	1.897216e-01	0.000e+00	2.311e+00	1.147e-01
7	33	1.513701e-01	0.000e+00	9.706e-01	5.764e-02
8	36	1.153330e-01	0.000e+00	1.127e+00	8.169e-02
9	39	1.198058e-01	0.000e+00	1.000e-01	1.522e-02
10	42	8.910052e-02	0.000e+00	8.378e-01	8.301e-02
11	45	6.771960e-02	0.000e+00	1.365e+00	7.149e-02
12	48	6.437664e-02	0.000e+00	1.146e-01	5.701e-03
13	51	6.329037e-02	0.000e+00	1.883e-02	3.774e-03
14	54	5.161934e-02	0.000e+00	3.016e-01	4.464e-02
15	57	4.964194e-02	0.000e+00	7.913e-02	7.894e-03
16	60	4.955404e-02	0.000e+00	5.462e-03	4.185e-04
17	63	4.954839e-02	0.000e+00	3.993e-03	2.208e-05
18	66	4.658289e-02	0.000e+00	1.318e-02	1.255e-02
19	69	4.647011e-02	0.000e+00	8.006e-04	4.940e-04
20	72	4.569141e-02	0.000e+00	3.136e-03	3.379e-03
21	75	4.568281e-02	0.000e+00	6.439e-05	3.974e-05
22	78	4.568281e-02	0.000e+00	8.000e-06	1.083e-07
23	81	4.567641e-02	0.000e+00	1.601e-06	2.793e-05
24	84	4.567482e-02	0.000e+00	2.062e-08	6.916e-06

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 24. `fmincon` took 24 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 84, indicating that `fmincon` evaluated Rosenbrock's function 84 times in the process of finding a minimum.
- The third column, labeled `f(x)`, displays the value of the objective function. The final value, 0.04567482, is the minimum that is reported in the Optimization app **Run solver and view results** box, and at the end of the exit message in the command window.
- The fourth column, `Feasibility`, is 0 for all iterations. This column shows the value of the constraint function `unitdisk` at each iteration where the constraint is positive. Since the value of `unitdisk` was negative in all iterations, every iteration satisfied the constraint.

The other columns of the iteration table are described in "Iterative Display" on page 3-16.

Set Up a Linear Program

In this section...
"Convert a Problem to Solver Form" on page 1-12
"Model Description" on page 1-12
"Solution Method" on page 1-13
"Bibliography" on page 1-20

Convert a Problem to Solver Form

This example shows how to convert a problem from mathematical form into Optimization Toolbox solver syntax. While the problem is a linear program, the techniques apply to all solvers.

The variables and expressions in the problem represent a model of operating a chemical plant, from an example in Edgar and Himmelblau [1]. There are two videos that describe the problem.

- Optimization Modeling 1 shows the problem in pictorial form. It shows how to generate the mathematical expressions of "Model Description" on page 1-12 from the picture.
- Optimization Modeling 2 describes how to convert these mathematical expressions into Optimization Toolbox solver syntax. This video shows how to solve the problem, and how to interpret the results.

The remainder of this example is concerned solely with transforming the problem to solver syntax. The example closely follows the video Optimization Modeling 2. The main difference between the video and the example is that this example shows how to use named variables, or index variables, which are similar to hash keys. This difference is in "Combine Variables Into One Vector" on page 1-15.

Model Description

The video Optimization Modeling 1 suggests that one way to convert a problem into mathematical form is to:

1   Get an overall idea of the problem

2 Identify the goal (maximizing or minimizing something)
3 Identify (name) variables
4 Identify constraints
5 Determine which variables you can control
6 Specify all quantities in mathematical notation
7 Check the model for completeness and correctness

For the meaning of the variables in this section, see the video Optimization Modeling 1.

The optimization problem is to minimize the objective function, subject to all the other expressions as constraints.

The objective function is:
```
0.002614 HPS + 0.0239 PP + 0.009825 EP.
```

The constraints are:
$2500 \leq P1 \leq 6250$
$I1 \leq 192,000$
$C \leq 62,000$
```
I1 - HE1 ≤ 132,000
```
```
I1 = LE1 + HE1 + C
```
```
1359.8 I1 = 1267.8 HE1 + 1251.4 LE1 + 192 C + 3413 P1
```
$3000 \leq P2 \leq 9000$
$I2 \leq 244,000$
$LE2 \leq 142,000$
```
I2 = LE2 + HE2
```
```
1359.8 I2 = 1267.8 HE2 + 1251.4 LE2 + 3413 P2
```
```
HPS = I1 + I2 + BF1
```
```
HPS = C + MPS + LPS
```
```
LPS = LE1 + LE2 + BF2
```
```
MPS = HE1 + HE2 + BF1 - BF2
```
$P1 + P2 + PP \geq 24,550$
$EP + PP \geq 12,000$
$MPS \geq 271,536$
$LPS \geq 100,623$
All variables are positive.

Solution Method

To solve the optimization problem, take the following steps.

The steps are also shown in the video Optimization Modeling 2.

Choose a Solver

To find the appropriate solver for this problem, consult the "Optimization Decision Table" on page 2-5. The table asks you to categorize your problem by type of objective function and types of constraints. For this problem, the objective function is linear, and the constraints are linear. The decision table recommends using the `linprog` solver.

As you see in "Problems Handled by Optimization Toolbox Functions" on page 2-14 or the `linprog` function reference page, the `linprog` solver solves problems of the form

$$\min_{x} f^{T}x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

- $f^{T}x$ means a row vector of constants f multiplying a column vector of variables x. In other words,
  $f^{T}x = f(1)x(1) + f(2)x(2) + ... + f(n)x(n),$

  where n is the length of f.
- $A x \leq b$ represents linear inequalities. A is a k-by-n matrix, where k is the number of inequalities and n is the number of variables (size of x). b is a vector of length k. For more information, see "Linear Inequality Constraints" on page 2-34.
- $Aeq x = beq$ represents linear equalities. Aeq is an m-by-n matrix, where m is the number of equalities and n is the number of variables (size of x). beq is a vector of length m. For more information, see "Linear Equality Constraints" on page 2-35.

- $lb \le x \le ub$ means each element in the vector x must be greater than the corresponding element of lb, and must be smaller than the corresponding element of ub. For more information, see "Bound Constraints" on page 2-33.

The syntax of the `linprog` solver, as shown in its function reference page, is

```
[x fval] = linprog(f,A,b,Aeq,beq,lb,ub);
```

The inputs to the `linprog` solver are the matrices and vectors in Equation 1-1.

Combine Variables Into One Vector

There are 16 variables in the equations of "Model Description" on page 1-12. Put these variables into one vector. The name of the vector of variables is x in Equation 1-1. Decide on an order, and construct the components of x out of the variables.

The following code constructs the vector using a cell array of strings. Each string is the name of a variable.

```
variables = {'I1','I2','HE1','HE2','LE1','LE2','C','BF1',...
    'BF2','HPS','MPS','LPS','P1','P2','PP','EP'};
N = length(variables);
% create variables for indexing
for v = 1:N
   eval([variables{v},' = ', num2str(v),';']);
end
```

Executing these commands creates the following named variables in your workspace:

These named variables represent index numbers for the components of *x*. You do not have to create named variables. The video Optimization Modeling 2 shows how to solve the problem simply using the index numbers of the components of *x*.

Write Bound Constraints

There are four variables with lower bounds, and six with upper bounds in the equations of "Model Description" on page 1-12. The lower bounds:

P1 ≥ 2500
P2 ≥ 3000
MPS ≥ 271,536
LPS ≥ 100,623.

Also, all the variables are positive, which means they have a lower bound of zero.

Create the lower bound vector `lb` as a vector of `0`, then add the four other lower bounds.

```
lb = zeros(size(variables));
lb([P1,P2,MPS,LPS]) = ...
    [2500,3000,271536,100623];
```

The variables with upper bounds are:
P1 $\leq$ 6250
P2 $\leq$ 9000
I1 $\leq$ 192,000
I2 $\leq$ 244,000
C $\leq$ 62,000
LE2 $\leq$ 142000.

Create the upper bound vector as a vector of `Inf`, then add the six upper bounds.

```
ub = Inf(size(variables));
ub([P1,P2,I1,I2,C,LE2]) = ...
    [6250,9000,192000,244000,62000,142000];
```

Write Linear Inequality Constraints

There are three linear inequalities in the equations of "Model Description" on page
1-12:
I1 - HE1 $\leq$ 132,000
EP + PP $\geq$ 12,000
P1 + P2 + PP $\geq$ 24,550.

In order to have the equations in the form $A\,x \leq b$, put all the variables on the left side of
the inequality. All these equations already have that form. Ensure that each inequality is
in "less than" form by multiplying through by –1 wherever appropriate:
I1 - HE1 $\leq$ 132,000
-EP - PP $\leq$ -12,000
-P1 - P2 - PP $\leq$ -24,550.

In your MATLAB workspace, create the A matrix as a 3-by-16 zero matrix, corresponding
to 3 linear inequalities in 16 variables. Create the b vector with three components.

```
A = zeros(3,16);
A(1,I1) = 1; A(1,HE1) = -1; b(1) = 132000;
A(2,EP) = -1; A(2,PP) = -1; b(2) = -12000;
A(3,[P1,P2,PP]) = [-1,-1,-1];
b(3) = -24550;
```

Write Linear Equality Constraints

There are eight linear equations in the equations of "Model Description" on page 1-12:
```
I2 = LE2 + HE2
LPS = LE1 + LE2 + BF2
HPS = I1 + I2 + BF1
HPS = C + MPS + LPS
I1 = LE1 + HE1 + C
MPS = HE1 + HE2 + BF1 - BF2
1359.8 I1 = 1267.8 HE1 + 1251.4 LE1 + 192 C + 3413 P1
1359.8 I2 = 1267.8 HE2 + 1251.4 LE2 + 3413 P2.
```

In order to have the equations in the form *Aeq x=beq*, put all the variables on one side of the equation. The equations become:
```
LE2 + HE2 - I2 = 0
LE1 + LE2 + BF2 - LPS = 0
I1 + I2 + BF1 - HPS = 0
C + MPS + LPS - HPS = 0
LE1 + HE1 + C - I1 = 0
HE1 + HE2 + BF1 - BF2 - MPS = 0
1267.8 HE1 + 1251.4 LE1 + 192 C + 3413 P1 - 1359.8 I1 = 0
1267.8 HE2 + 1251.4 LE2 + 3413 P2 - 1359.8 I2 = 0.
```

Now write the `Aeq` matrix and `beq` vector corresponding to these equations. In your MATLAB workspace, create the `Aeq` matrix as an 8-by-16 zero matrix, corresponding to 8 linear equations in 16 variables. Create the `beq` vector with eight components, all zero.

```
Aeq = zeros(8,16); beq = zeros(8,1);
Aeq(1,[LE2,HE2,I2]) = [1,1,-1];
Aeq(2,[LE1,LE2,BF2,LPS]) = [1,1,1,-1];
Aeq(3,[I1,I2,BF1,HPS]) = [1,1,1,-1];
Aeq(4,[C,MPS,LPS,HPS]) = [1,1,1,-1];
Aeq(5,[LE1,HE1,C,I1]) = [1,1,1,-1];
Aeq(6,[HE1,HE2,BF1,BF2,MPS]) = [1,1,1,-1,-1];
Aeq(7,[HE1,LE1,C,P1,I1]) = [1267.8,1251.4,192,3413,-1359.8];
Aeq(8,[HE2,LE2,P2,I2]) = [1267.8,1251.4,3413,-1359.8];
```

Write the Objective

The objective function is
$f^Tx = 0.002614$ HPS $+ 0.0239$ PP $+ 0.009825$ EP.

Write this expression as a vector `f` of multipliers of the *x* vector:

```
f = zeros(size(variables));
f([HPS PP EP]) = [0.002614 0.0239 0.009825];
```

Solve the Problem with linprog

You now have inputs required by the `linprog` solver. Call the solver and print the outputs in formatted form:

```
[x fval] = linprog(f,A,b,Aeq,beq,lb,ub);
for d = 1:N
  fprintf('%12.2f \t%s\n',x(d),variables{d})
end
fval
```

The result:

```
Optimization terminated.
    136328.74   I1
    244000.00   I2
    128159.00   HE1
    143377.00   HE2
         0.00   LE1
    100623.00   LE2
      8169.74   C
         0.00   BF1
         0.00   BF2
    380328.74   HPS
    271536.00   MPS
    100623.00   LPS
      6250.00   P1
      7060.71   P2
     11239.29   PP
       760.71   EP

fval =
  1.2703e+003
```

Examine the Solution

The `fval` output gives the smallest value of the objective function at any feasible point.

The solution vector `x` is the point where the objective function has the smallest value. Notice that:

- BF1, BF2, and LE1 are 0, their lower bounds.
- I2 is 244,000, its upper bound.
- The nonzero components of the f vector are

  - HPS — 380,328.74
  - PP — 11,239.29
  - EP — 760.71

The video Optimization Modeling 2 gives interpretations of these characteristics in terms of the original problem.

Bibliography

[1] Edgar, Thomas F., and David M. Himmelblau. *Optimization of Chemical Processes.* McGraw-Hill, New York, 1988.

2

Setting Up an Optimization

Optimization Theory Overview

Optimization techniques are used to find a set of design parameters, $x = \{x_1, x_2, ..., x_n\}$, 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

$$\min_x f(x),$$

subject to

$$G_i(x) = 0 \quad i = 1, ..., m_e,$$
$$G_i(x) \leq 0 \quad i = m_e + 1, ..., m,$$

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.

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 2-14. See "Optimization Decision Table" on page 2-5 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 2-28.

Write the objective function for a solver in the form of a function 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 2-17. Constraints have a special form, as described in "Writing Constraints" on page 2-31.

Choosing a Solver

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 2-14.

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:

  - * means relevant solvers are found in Global Optimization 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` can be effective on some nonsmooth problems.

- The Global Optimization Toolbox `ga` function can address mixed-integer programming 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, Information	\, lsqcurvefit, lsqnonlin, Information	fminsearch, fminunc, Information	fminsearch, *
Bound	linprog, Information	quadprog, Information	lsqcurvefit, lsqlin, lsqnonlin, lsqnonneg, Information	fminbnd, fmincon, fseminf, Information	fminbnd, *
Linear	linprog, Information	quadprog, Information	lsqlin, Information	fmincon, fseminf, Information	*
General smooth	fmincon, Information	fmincon, Information	fmincon, Information	fmincon, fseminf, Information	*
Discrete	intlinprog, Information	*	*	*	*

Note: This table does not list multiobjective solvers nor equation solvers. See "Problems Handled by Optimization Toolbox Functions" on page 2-14 for a complete list of problems addressed by Optimization Toolbox functions.

> **Note:** Some solvers have several algorithms. For help choosing, see "Choosing the Algorithm" on page 2-7.

Choosing the Algorithm

fmincon Algorithms

`fmincon` has four algorithm options:

- `'interior-point'` (default)
- `'trust-region-reflective'`
- `'sqp'`
- `'active-set'`

Use `optimoptions` to set the `Algorithm` option at the command line.

Recommendations
• Use the `'interior-point'` algorithm first. For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15. • To run an optimization again to obtain more speed on small- to medium-sized problems, try `'sqp'` next, and `'active-set'` last. • Use `'trust-region-reflective'` when applicable. Your problem must have: objective function includes gradient, only bounds, or only linear equality constraints (but not both). See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-12.

Reasoning Behind the Recommendations

- `'interior-point'` handles large, sparse problems, as well as small dense problems. The algorithm satisfies bounds at all iterations, and can recover from NaN or Inf results. It is a large-scale algorithm; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12. The algorithm can use special techniques for large-scale problems. For details, see **Interior-Point Algorithm** in fmincon options.

- `'sqp'` satisfies bounds at all iterations. The algorithm can recover from NaN or Inf results. It is not a large-scale algorithm; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12.

- `'active-set'` can take large steps, which adds speed. The algorithm is effective on some problems with nonsmooth constraints. It is not a large-scale algorithm; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12.

- `'trust-region-reflective'` requires you to provide a gradient, and allows only bounds or linear equality constraints, but not both. Within these limitations, the algorithm handles both large sparse problems and small dense problems efficiently. It is a large-scale algorithm; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12. The algorithm can use special techniques to save memory usage, such as a Hessian multiply function. For details, see **Trust-Region-Reflective Algorithm** in fmincon options.

fsolve Algorithms

fsolve has three algorithms:

- `'trust-region-dogleg'` (default)
- `'trust-region-reflective'`
- `'levenberg-marquardt'`

Use optimoptions to set the Algorithm option at the command line.

Recommendations
• Use the `'trust-region-dogleg'` algorithm first.   For help if fsolve fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.   • To solve equations again if you have a Jacobian multiply function, or want to tune the internal algorithm (see **Trust-Region-Reflective Algorithm** in fsolve options), try `'trust-region-reflective'`.

> **Recommendations**
>
> - Try timing all the algorithms, including `'levenberg-marquardt'`, to find the algorithm that works best on your problem.

Reasoning Behind the Recommendations

- `'trust-region-dogleg'` is the only algorithm that is specially designed to solve nonlinear equations. The others attempt to minimize the sum of squares of the function.
- The `'trust-region-reflective'` algorithm is effective on sparse problems. It can use special techniques such as a Jacobian multiply function for large-scale problems.

fminunc Algorithms

fminunc has two algorithms:

- `'trust-region'` (formerly LargeScale = `'on'`), the default
- `'quasi-newton'` (formerly LargeScale = `'off'`)

Use optimoptions to set the Algorithm option at the command line.

> **Recommendations**
>
> - If your objective function includes a gradient, use `'Algorithm'` = `'trust-region'`, and set the GradObj option to `'on'`.
> - Otherwise, use `'Algorithm'` = `'quasi-newton'`.
>
> For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.

Least Squares Algorithms

lsqlin

lsqlin has three algorithms:

- `'trust-region-reflective'` (formerly LargeScale = `'on'`), the default
- `'active-set'` (formerly LargeScale = `'off'`)

- `'interior-point'`

Use `optimoptions` to set the `Algorithm` option at the command line.

Recommendations

- If you have no constraints or only bound constraints, use `'trust-region-reflective'`.
- If you have linear constraints, try `'interior-point'` first.

For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.

---

lsqcurvefit and lsqnonlin

`lsqcurvefit` and `lsqnonlin` have two algorithms:

- `'trust-region-reflective'` (default)
- `'levenberg-marquardt'`

Use `optimoptions` to set the `Algorithm` option at the command line.

Recommendations

- If you have no constraints or only bound constraints, use `'trust-region-reflective'`.
- If your problem is underdetermined (fewer equations than dimensions), use `'levenberg-marquardt'`.

For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.

Linear Programming Algorithms

`linprog` has five algorithms:

- `'interior-point-legacy'`, the default
- `'interior-point'`

- `'dual-simplex'`
- `'simplex'`
- `'active-set'`

Use `optimoptions` to set the `Algorithm` option at the command line.

Recommendations

Use the `'interior-point'` algorithm or the `'dual-simplex'` algorithm first.

For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.

See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-12.

Reasoning Behind the Recommendations

- The `'interior-point-legacy'`, `'interior-point'` and `'dual-simplex'` algorithms are large-scale algorithms, while `'simplex'` and `'active-set'` are not. See "Large-Scale vs. Medium-Scale Algorithms" on page 2-12.
- Often, the `'interior-point'` and `'dual-simplex'` algorithms are faster and use less memory than the other algorithms.
- The default `'interior-point-legacy'` is similar to `'interior-point'`, but `'interior-point-legacy'` can be slower, less robust, or use more memory.
- The `'active-set'` and `'simplex'` algorithms will be removed in a future release.

Quadratic Programming Algorithms

`quadprog` has three algorithms:

- `'interior-point-convex'` (default)
- `'trust-region-reflective'`
- `'active-set'` (will be removed in a future release).

Use `optimoptions` to set the `Algorithm` option at the command line.

Recommendations

- If you have a convex problem, or if you don't know whether your problem is convex, use `'interior-point-convex'`.

Recommendations

- If you have a nonconvex problem with only bounds, or with only linear equalities, use `'trust-region-reflective'`.

For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-15.

See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-12.

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.

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.

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.

Potential Inaccuracy with Interior-Point Algorithms

Interior-point algorithms in `fmincon`, `quadprog`, and `linprog` have many good characteristics, such as low memory usage and the ability to solve large problems quickly. However, their solutions can be slightly less accurate than those from other algorithms. The reason for this potential inaccuracy is that the (internally calculated) barrier function keeps iterates away from inequality constraint boundaries.

For most practical purposes, this inaccuracy is usually quite small.

To reduce the inaccuracy, try to:

- Rerun the solver with smaller `TolX`, `TolFun`, and possibly `TolCon` tolerances (but keep the tolerances sensible.) See "Tolerances and Stopping Criteria" on page 2-61).

- Run a different algorithm, starting from the interior-point solution. This can fail, because some algorithms can use excessive memory or time, and some `linprog` and `quadprog` algorithms do not accept an initial point.

For example, try to minimize the function x when bounded below by 0. Using the `fmincon` `interior-point` algorithm:

```
options = optimoptions(@fmincon,'Algorithm','interior-point','Display','off');
x = fmincon(@(x)x,1,[],[],[],[],0,[],[],options)

x =

   2.0000e-08
```

Using the `fmincon` `sqp` algorithm:

```
options.Algorithm = 'sqp';
x2 = fmincon(@(x)x,1,[],[],[],[],0,[],[],options)

x2 =

   0
```

Similarly, solve the same problem using the `linprog` `interior-point` algorithm:

```
opts = optimoptions(@linprog,'Display','off','Algorithm','interior-point');
x = linprog(1,[],[],[],[],0,[],1,opts)

x =

   2.0833e-13
```

Using the `linprog` `simplex` algorithm:

```
opts.Algorithm = 'simplex';
x2 = linprog(1,[],[],[],[],0,[],1,opts)

x2 =

    0
```

In these cases, the `interior-point` algorithms are less accurate, but the answers are quite close to the correct answer.

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	$\min\limits_{x} f(x)$ such that $lb < x < ub$ (x is scalar)	`fminbnd`
Unconstrained minimization	$\min\limits_{x} f(x)$	`fminunc,` `fminsearch`
Linear programming	$\min\limits_{x} f^{T}x$ such that $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$	`linprog`
Mixed-integer linear programming	$\min\limits_{x} f^{T}x$ such that $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$, x(intcon) is integer-valued.	`intlinprog`
Quadratic programming	$\min\limits_{x} \dfrac{1}{2}x^{T}Hx + c^{T}x$ such that $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$	`quadprog`
Constrained minimization	$\min\limits_{x} f(x)$ such that $c(x) \le 0$, $ceq(x) = 0$, $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$	`fmincon`
Semi-infinite minimization	$\min\limits_{x} f(x)$ such that $K(x,w) \le 0$ for all w, $c(x) \le 0$, $ceq(x) = 0$, $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$	`fseminf`

Multiobjective Problems

Type	Formulation	Solver
Goal attainment	$$\min_{x,\gamma} \gamma$$ such that $F(x) - w\,\gamma \leq$ goal, $c(x) \leq 0$, $ceq(x) = 0$, $A\,x \leq b$, $Aeq\,x = beq$, $lb \leq x \leq ub$	`fgoalattain`
Minimax	$$\min_{x} \max_{i} F_i(x)$$ such that $c(x) \leq 0$, $ceq(x) = 0$, $A\,x \leq b$, $Aeq\,x = beq$, $lb \leq x \leq ub$	`fminimax`

Equation Solving Problems

Type	Formulation	Solver
Linear equations	$C\,x = d$, 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	$$\min_{x} \frac{1}{2}\|C \cdot x - d\|_2^2$$ m equations, n variables	\ (matrix left division)
Nonnegative linear-least-squares	$$\min_{x} \frac{1}{2}\|C \cdot x - d\|_2^2$$ such that $x \geq 0$	`lsqnonneg`
Constrained linear-least-squares	$$\min_{x} \frac{1}{2}\|C \cdot x - d\|_2^2$$	`lsqlin`

Type	Formulation	Solver
	such that $A\,x \leq b$, $Aeq\,x = beq$, $lb \leq x \leq ub$	
Nonlinear least-squares	$$\min_x \|F(x)\|_2^2 = \min_x \sum_i F_i^2(x)$$ such that $lb \leq x \leq ub$	`lsqnonlin`
Nonlinear curve fitting	$$\min_x \|F(x,xdata) - ydata\|_2^2$$ such that $lb \leq x \leq ub$	`lsqcurvefit`

Writing Objective Functions

In this section...
"Types of Objective Functions" on page 2-17
"Writing Scalar Objective Functions" on page 2-18
"Writing Vector and Matrix Objective Functions" on page 2-25
"Writing Objective Functions for Linear or Quadratic Problems" on page 2-28
"Maximizing an Objective" on page 2-28

Types of Objective Functions

Many Optimization Toolbox solvers minimize a scalar function of a multidimensional vector. The *objective* function is the function the solvers attempt to minimize. Several solvers accept vector-valued objective functions, and some solvers use objective functions you specify by vectors or matrices.

Objective Type	Solvers	How to Write Objectives

Scalar	`fmincon`
	
	`fminunc`
	
	`fminbnd`
	
	`fminsearch`
	
	`fseminf`
	
	`fzero`
		"Writing Scalar Objective Functions" on page 2-18

Nonlinear least squares	`lsqcurvefit`
	
	`lsqnonlin`
		"Writing Vector and Matrix Objective Functions" on page 2-25

Multivariable equation solving	`fsolve`	

Multiobjective	`fgoalattain`
	
	`fminimax`	

Objective Type	Solvers	How to Write Objectives
Linear programming	`linprog`	"Writing Objective Functions for Linear or Quadratic Problems" on page 2-28
Mixed-integer linear programming	`intlinprog`	

Linear least squares	`lsqlin`
	
	`lsqnonneg`	

Quadratic programming	`quadprog`	

Writing Scalar Objective Functions

- "Function Files" on page 2-18
- "Anonymous Function Objectives" on page 2-20
- "Including Derivatives" on page 2-20

Function Files

A scalar objective function file accepts one input, say x, and returns one scalar output, say f. The input x can be a scalar, vector, or matrix. A function file can return more outputs (see "Including Derivatives" on page 2-20).

For example, suppose your objective is a function of three variables, x, y, and z:
$f(x) = 3*(x - y)^4 + 4*(x + z)^2 / (1 + x^2 + y^2 + z^2) + \cosh(x - 1) + \tanh(y + z)$.

1 Write this function as a file that accepts the vector xin = [x;y;z] and returns f:

```
function f = myObjective(xin)
f = 3*(xin(1)-xin(2))^4 + 4*(xin(1)+xin(3))^2/(1+norm(xin)^2) ...
    + cosh(xin(1)-1) + tanh(xin(2)+xin(3));
```
2 Save it as a file named `myObjective.m` to a folder on your MATLAB path.
3 Check that the function evaluates correctly:

```
myObjective([1;2;3])

ans =
    9.2666
```

For information on how to include extra parameters, see "Passing Extra Parameters" on page 2-50. For more complex examples of function files, see "Minimization with

Gradient and Hessian Sparsity Pattern" on page 6-17 or "Minimization with Bound Constraints and Banded Preconditioner" on page 6-68.

Local Functions and Nested Functions

Functions can exist inside other files as local functions or nested functions. Using local functions or nested functions can lower the number of distinct files you save. Using nested functions also lets you access extra parameters, as shown in "Nested Functions" on page 2-52.

For example, suppose you want to minimize the myObjective.m objective function, described in "Function Files" on page 2-18, subject to the ellipseparabola.m constraint, described in "Nonlinear Constraints" on page 2-35. Instead of writing two files, myObjective.m and ellipseparabola.m, write one file that contains both functions as local functions:

```
function [x fval] = callObjConstr(x0,options)
% Using a local function for just one file

if nargin < 2
    options = optimoptions('fmincon','Algorithm','interior-point');
end

[x fval] = fmincon(@myObjective,x0,[],[],[],[],[],[], ...
    @ellipseparabola,options);

function f = myObjective(xin)
f = 3*(xin(1)-xin(2))^4 + 4*(xin(1)+xin(3))^2/(1+sum(xin.^2)) ...
    + cosh(xin(1)-1) + tanh(xin(2)+xin(3));

function [c,ceq] = ellipseparabola(x)
c(1) = (x(1)^2)/9 + (x(2)^2)/4 - 1;
c(2) = x(1)^2 - x(2) - 1;
ceq = [];
```

Solve the constrained minimization starting from the point [1;1;1]:

```
[x fval] = callObjConstr(ones(3,1))

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is
non-decreasing in feasible directions, to within the default
value of the function tolerance, and constraints are satisfied
```

```
to within the default value of the constraint tolerance.

x =
    1.1835
    0.8345
   -1.6439

fval =
    0.5383
```

Anonymous Function Objectives

Use anonymous functions to write simple objective functions. For more information about anonymous functions, see "What Are Anonymous Functions?" in the MATLAB Programming Fundamentals documentation. 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 `anonrosen` evaluates correctly at `[-1 2]`:

```
anonrosen([-1 2])

ans =
    104
```
Minimizing `anonrosen` with `fminunc` yields the following results:

```
options = optimoptions(@fminunc,'Algorithm','quasi-newton');
[x fval] = fminunc(anonrosen,[-1;2],options)

Local minimum found.

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

x =
    1.0000
    1.0000

fval =
  1.2262e-10
```

Including Derivatives

For `fmincon` and `fminunc`, you can include gradients in the objective function. You can also include Hessians, depending on the algorithm. The Hessian matrix $H_{i,j}(x) = \partial^2 f / \partial x_i \partial x_j$.

The following table shows which algorithms can use gradients and Hessians.

Solver	Algorithm	Gradient	Hessian
fmincon	active-set	Optional	No
	interior-point	Optional	Optional separate function (see "Hessian" on page 14-60)
	sqp	Optional	No
	trust-region-reflective	Required	Optional
fminunc	trust-region	Required	Optional
	quasi-newton	Optional	No

- "Benefits of Including Derivatives" on page 2-21
- "Choose Input Hessian for interior-point fmincon" on page 2-21
- "How to Include Derivatives" on page 2-22

Benefits of Including Derivatives

If you do not provide gradients, solvers estimate gradients via finite differences. If you provide gradients, your solver need not perform this finite difference estimation, so can save time and be more accurate. Furthermore, solvers use an approximate Hessian, which can be far from the true Hessian. Providing a Hessian can yield a solution in fewer iterations.

For constrained problems, providing a gradient has another advantage. A solver can reach a point x such that x is feasible, but, for this x, finite differences around x always lead to an infeasible point. Suppose further that the objective function at an infeasible point returns a complex output, Inf, NaN, or error. In this case, a solver can fail or halt prematurely. Providing a gradient allows a solver to proceed. To obtain this benefit, you might also need to include the gradient of a nonlinear constraint function, and set the GradConstr option to 'on'. See "Nonlinear Constraints" on page 2-35.

Choose Input Hessian for interior-point fmincon

The fmincon interior-point algorithm has many options for selecting an input Hessian. For syntax details, see "Hessian" on page 14-60. Here are the options, along with estimates of their relative characteristics.

Hessian	Relative Memory Usage	Relative Efficiency
'bfgs' (default)	High (for large problems)	High

Hessian	Relative Memory Usage	Relative Efficiency
`'lbfgs'`	Low to Moderate	Moderate
`'fin-diff-grads'`	Low	Moderate
`'user-supplied'` with `'HessMult'`	Low (can depend on your code)	Moderate
`'user-supplied'` with `'HessFcn'`	? (depends on your code)	High (depends on your code)

Use the default `'bfgs'` Hessian unless you

- Run out of memory — Try `'lbfgs'` instead of `'bfgs'`. If you can provide your own gradients, try `'fin-diff-grads'`, and set the GradObj and GradConstr options to `'on'`.
- Want more efficiency — Provide your own gradients and Hessian. See "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50 and "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

The reason `'lbfgs'` has only moderate efficiency is twofold. It has relatively expensive Sherman-Morrison updates. And the resulting iteration step can be somewhat inaccurate due to the `'lbfgs'` limited memory.

The reason `'fin-diff-grads'` and HessMult have only moderate efficiency is that they use a conjugate gradient approach. They accurately estimate the Hessian of the objective function, but they do not generate the most accurate iteration step. For more information, see "fmincon Interior Point Algorithm" on page 6-37, and its discussion of the LDL approach and the conjugate gradient approach to solving Equation 6-52.

How to Include Derivatives

1 Write code that returns:

- The objective function (scalar) as the first output
- The gradient (vector) as the second output
- Optionally, the Hessian (matrix) as the third output

2 Set the GradObj option to `'on'` with optimoptions.

3 Optionally, set the Hessian option to `'on'` or `'user-supplied'`.

   For the fmincon interior-point solver, set the Hessian option to `'user-supplied'` and set the `'HessFcn'` option to @*hessianfcn*, where *hessianfcn* is a

function that computes the Hessian of the Lagrangian. For details, see "Hessian" on page 14-60. For an example, see "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50.

4 Optionally, check if your gradient function matches a finite-difference approximation. See "Checking Validity of Gradients or Jacobians" on page 2-65.

Tip For most flexibility, write *conditionalized* code. Conditionalized means that the number of function outputs can vary, as shown in the following example. Conditionalized code does not error depending on the value of the `GradObj` or `Hessian` option. Unconditionalized code requires you to set these options appropriately.

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 "Solve a Constrained Nonlinear Problem" on page 1-3. The gradient of $f(x)$ is

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

and the Hessian $H(x)$ is

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

`rosenthree` is an unconditionalized function that returns the Rosenbrock function with its gradient and Hessian:

```
function [f g H] = rosenthree(x)
% Calculate objective f, gradient g, Hessian H
f = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;
g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
        200*(x(2)-x(1)^2)];
H = [1200*x(1)^2-400*x(2)+2, -400*x(1);
```

```
                -400*x(1), 200];
```

`rosenboth` is a conditionalized function that returns whatever the solver requires:

```
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 "Find Number of Function Arguments" in the MATLAB Programming Fundamentals documentation.

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

```
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'GradObj','on','Hessian','on');
```

Run `fminunc` starting at `[-1;2]`:

```
[x fval] = fminunc(@rosenboth,[-1;2],options)
Local minimum found.

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

x =
    1.0000
    1.0000

fval =
  1.9310e-017
```

If you have a Symbolic Math Toolbox™ license, you can calculate gradients and Hessians automatically, as described in "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

Writing Vector and Matrix Objective Functions

Some solvers, such as `fsolve` and `lsqcurvefit`, have objective functions that are vectors or matrices. The main 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" on page 2-25
- "Jacobians of Matrix Functions" on page 2-26
- "Jacobians with Matrix-Valued Independent Variables" on page 2-27

Jacobians of Vector Functions

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

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

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

For example, if

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

then $J(x)$ is

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

The function file associated with this example is:

```
function [F jacF] = vectorObjective(x)
F = [x(1)^2 + x(2)*x(3);
```

```
      sin(x(1) + 2*x(2) - 3*x(3))];
if nargout > 1 % need Jacobian
    jacF = [2*x(1),x(3),x(2);
        cos(x(1)+2*x(2)-3*x(3)),2*cos(x(1)+2*x(2)-3*x(3)), ...
        -3*cos(x(1)+2*x(2)-3*x(3))];
end
```

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, rewrite the matrix

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

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 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 an mn-by-k matrix.

For example, if

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

then 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, define the Jacobian of *F*(*x*) 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 with *f* the vector form of *F* as above, the Jacobian of *F*(*X*) is defined as the Jacobian of *f*(*x*):

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

2-27

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, then the Jacobian is an mn-by-jk matrix.

Writing Objective Functions for Linear or Quadratic Problems

The following solvers handle linear or quadratic objective functions:

- `linprog` and `intlinprog`: minimize
  `f'x = f(1)*x(1) + f(2)*x(2) +...+ f(n)*x(n)`.

  Input the vector `f` for the objective. See the examples in "Linear Programming and Mixed-Integer Linear Programming".

- `lsqlin` and `lsqnonneg`: minimize
  `‖Cx - d‖`.

  Input the matrix `C` and the vector `d` for the objective. See "Linear Least Squares with Bound Constraints" on page 10-21.

- `quadprog`: minimize
  `1/2 * x'Hx + f'x`
  `= 1/2 * (x(1)*H(1,1)*x(1) + 2*x(1)*H(1,2)*x(2) +...`
  `+ x(n)*H(n,n)*x(n)) + f(1)*x(1) + f(2)*x(2) +...+ f(n)*x(n)`.

  Input both the vector `f` and the symmetric matrix `H` for the objective. See "Quadratic Programming".

Maximizing an Objective

All solvers attempt 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)
Local minimum found.

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

x =
    6.2832

fval =
    -1.5574
```

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

Matrix Arguments

Solvers accept matrix initial point `x0`, where matrix means an array of any size. They also accept matrix bounds `lb` and `ub`. Here's how solvers handle matrix arguments.

- Internally, solvers convert matrix arguments into vectors before processing. For example, `x0` becomes `x0(:)`. For an explanation of this syntax, see the `A(:)` entry in `colon`.
- For output, solvers reshape the solution `x` to the same size as the input `x0`.
- When `x0` is a matrix, solvers pass `x` as a matrix of the same size as `x0` to both the objective function and to any nonlinear constraint function.
- Linear constraints, though, take `x` in vector form, `x(:)`. In other words, a linear constraint of the form
  `A*x ≤ b` or `Aeq*x = beq`

  takes `x` as a vector, not a matrix. Ensure that your matrix `A` or `Aeq` has the same number of columns as `x0` has elements, or the solver will error.

Writing Constraints

In this section...

Types of Constraints

Optimization Toolbox solvers have special forms for constraints:

- "Bound Constraints" on page 2-33 — Lower and upper bounds on individual components: $x \geq l$ and $x \leq u$.
- "Linear Inequality Constraints" on page 2-34 — $A\,x \leq b$. 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 2-35 — $Aeq\,x = beq$. Equality constraints have the same form as inequality constraints.
- "Nonlinear Constraints" on page 2-35 — $c(x) \leq 0$ 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 $A\,x \leq b$. Express greater-than constraints 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 $A\,x \geq b$ is equivalent to the constraint $-A\,x \leq -b$. For more information, see "Linear Inequality Constraints" on page 2-34 and "Nonlinear Constraints" on page 2-35.

You can sometimes write constraints in several ways. For best results, 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 2-50.

Iterations Can Violate Constraints

Be careful when writing your objective and constraint functions. Intermediate iterations can lead to points that are infeasible (do not satisfy constraints). If you write objective or constraint functions that assume feasibility, these functions can error or give unexpected results.

For example, if you take a square root or logarithm of x, and $x < 0$, the result is not real. You can try to avoid this error by setting 0 as a lower bound on x. Nevertheless, an intermediate iteration can violate this bound.

Algorithms That Satisfy Bound Constraints

Some solver algorithms satisfy bound constraints at every iteration:

- `fmincon interior-point`, `sqp`, and `trust-region-reflective` algorithms
- `lsqcurvefit trust-region-reflective` algorithm
- `lsqnonlin trust-region-reflective` algorithm
- `fminbnd`

Note: If you set a lower bound equal to an upper bound, iterations can violate these constraints.

Solvers and Algorithms That Can Violate Bound Constraints

The following solvers and algorithms can violate bound constraints at intermediate iterations:

- fmincon active-set algorithm
- fgoalattain solver
- fminimax solver
- fseminf solver

Bound Constraints

Lower and upper bounds limit the components of the solution x.

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

Give bounds as vectors with the same length as x, or as matrices with the same number of elements as x.

- If a particular component has no lower bound, use -Inf as the bound; similarly, use Inf if a component has no upper bound.
- 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 Infs.
- If only the first m out of n components have bounds, then you need only supply a vector of length m containing bounds. However, this shortcut causes solvers to throw a warning.

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] (throws a warning) or u = [Inf; 3; Inf].

Tip Use Inf or -Inf instead of a large, arbitrary bound to lower memory usage and increase solver speed. See "Use Inf Instead of a Large, Arbitrary Bound" on page 4-13.

You need not give gradients for bound constraints; solvers calculate them automatically. Bounds do not affect Hessians.

For a more complex example of bounds, see "Set Up a Linear Program" on page 1-12.

Linear Inequality Constraints

Linear inequality constraints have the form $A\,x \le b$. When A is m-by-n, there are m constraints on a variable x with n components. You supply the m-by-n matrix A and the m-component vector b.

Even if you pass an initial point x0 as a matrix, solvers pass the current point x as a column vector to linear constraints. See "Matrix Arguments" on page 2-30.

For example, suppose that you have the following linear inequalities as constraints:
$x_1 + x_3 \le 4$,
$2x_2 - x_3 \ge -2$,
$x_1 - x_2 + x_3 - x_4 \ge 9$.

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. In MATLAB syntax:

```
A = [1 0 1 0;
     0 -2 1 0;
     -1 1 -1 1];
b = [4;2;-9];
```

You do not need to give gradients for linear constraints; solvers calculate them automatically. Linear constraints do not affect Hessians.

For a more complex example of linear constraints, see "Set Up a Linear Program" on page 1-12.

Linear Equality Constraints

Linear equalities have the form *Aeq x = beq*, which 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 gradients for linear constraints; solvers calculate them automatically. Linear constraints do not affect Hessians. The form of this type of constraint is the same as for "Linear Inequality Constraints" on page 2-34.

Nonlinear Constraints

Nonlinear inequality constraints have the form $c(x) \leq 0$, where *c* is a vector of constraints, one component for each constraint. Similarly, nonlinear equality constraints are of the form $ceq(x) = 0$.

Note: Nonlinear constraint functions must return both `c` and `ceq`, the inequality and equality constraint functions, even if they do not both exist. Return empty `[]` for a nonexistent constraint.

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

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

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

Write these constraints in a function file as follows:

```
function [c,ceq]=ellipseparabola(x)
c(1) = (x(1)^2)/9 + (x(2)^2)/4 - 1;
c(2) = x(1)^2 - x(2) - 1;
ceq = [];
end
```

`ellipseparabola` returns empty `[]` for `ceq`, the nonlinear equality function. Also, both inequalities were put into ≤ 0 form.

Including Gradients in Constraint Functions

If you provide gradients for *c* and *ceq*, your solver can run faster and give more reliable results.

Providing a gradient has another advantage. A solver can reach a point x such that x is feasible, but finite differences around x always lead to an infeasible point. In this case, a solver can fail or halt prematurely. Providing a gradient allows a solver to proceed.

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

```
function [c,ceq,gradc,gradceq]=ellipseparabola(x)
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 Scalar Objective Functions" on page 2-18 for information on conditionalized functions. The gradient matrix has the form $\text{gradc}_{i,j} = [\partial\text{c}(j)/\partial x_i]$.

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, indicate that they exist by using optimoptions:

```
options=optimoptions(@fmincon,'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)
```

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians automatically, as described in "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

Anonymous Nonlinear Constraint Functions

For information on anonymous objective functions, see "Anonymous Function Objectives" on page 2-20.

Nonlinear constraint functions must return two outputs. The first output corresponds to nonlinear inequalities, and the second corresponds to nonlinear equalities.

Anonymous functions return just one output. So how can you write an anonymous function as a nonlinear constraint?

The `deal` function distributes multiple outputs. For example, suppose your nonlinear inequalities are

$$\frac{x_1^2}{9} + \frac{x_2^2}{4} \leq 1,$$
$$x_2 \geq x_1^2 - 1.$$

Suppose that your nonlinear equality is
$x_2 = \tanh(x_1)$.

Write a nonlinear constraint function as follows:

```
c = @(x)[x(1)^2/9 + x(2)^2/4 - 1;
        x(1)^2 - x(2) - 1];
ceq = @(x)tanh(x(1)) - x(2);
nonlinfcn = @(x)deal(c(x),ceq(x));
```

To minimize the function $\cosh(x_1) + \sinh(x_2)$ subject to the constraints in `nonlinfcn`, use `fmincon`:

```
obj = @(x)cosh(x(1))+sinh(x(2));
opts = optimoptions(@fmincon,'Algorithm','sqp');
z = fmincon(obj,[0;0],[],[],[],[],[],[],nonlinfcn,opts)

Local minimum found that satisfies the constraints.

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

z =
   -0.6530
   -0.5737
```

To check how well the resulting point `z` satisfies the constraints, use `nonlinfcn`:

```
[cout,ceqout] = nonlinfcn(z)

cout =
```

```
   -0.8704
         0

ceqout =
    1.1102e-016
```

z indeed satisfies all the constraints to within the default value of the `TolCon` constraint tolerance, `1e-6`.

Or Instead of And Constraints

In general, solvers takes constraints with an implicit AND:

constraint 1 AND constraint 2 AND constraint 3 are all satisfied.

However, sometimes you want an OR:

constraint 1 OR constraint 2 OR constraint 3 is satisfied.

These formulations are not logically equivalent, and there is generally no way to express OR constraints in terms of AND constraints.

Tip Fortunately, nonlinear constraints are extremely flexible. You get OR constraints simply by setting the nonlinear constraint function to the minimum of the constraint functions.

The reason that you can set the minimum as the constraint is due to the nature of "Nonlinear Constraints" on page 2-35: you give them as a set of functions that must be negative at a feasible point. If your constraints are
$F_1(x) \le 0$ OR $F_2(x) \le 0$ OR $F_3(x) \le 0$,

then set the nonlinear inequality constraint function $c(x)$ as:
$c(x) = \min(F_1(x), F_2(x), F_3(x))$.

$c(x)$ is not smooth, which is a general requirement for constraint functions, due to the minimum. Nevertheless, the method often works.

Note: You cannot use the usual bounds and linear constraints in an OR constraint. Instead, convert your bounds and linear constraints to nonlinear constraint functions, as in this example.

For example, suppose your feasible region is the L-shaped region: x is in the rectangle $-1 \le x(1) \le 1$, $0 \le x(2) \le 1$ OR x is in the rectangle $0 \le x(1) \le 1$, $-1 \le x(2) \le 1$.
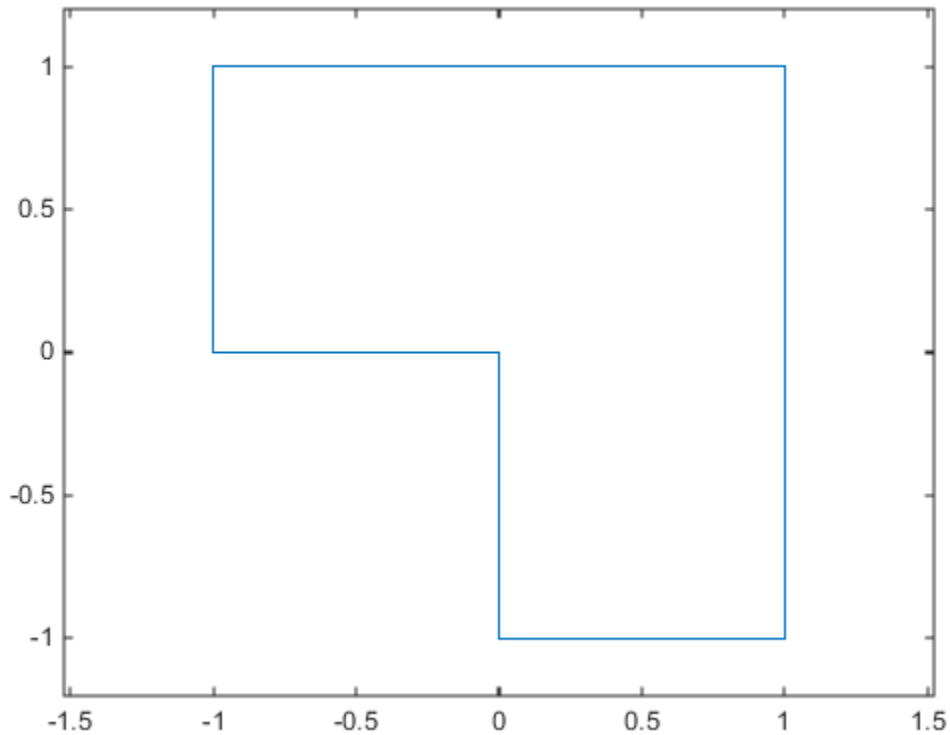
Code for creating the figure

```
% Write the x and y coordinates of the figure, clockwise from (0,0)
x = [0,-1,-1,1,1,0,0];
y = [0,0,1,1,-1,-1,0];
plot(x,y)
xlim([-1.2 1.2])
ylim([-1.2 1.2])
axis equal
```

To represent a rectangle as a nonlinear constraint, instead of as bound constraints, construct a function that is negative inside the rectangle $a \leq x(1) \leq b$, $c \leq x(2) \leq d$:

```
function cout = rectconstr(x,a,b,c,d)
% Negative when  x is in the rectangle [a,b][c,d]
% First check that a,b,c,d are in the correct order

if (b <= a) || (d <= c)
    error('Give a rectangle a < b, c < d')
end

cout = max([(x(1)-b),(x(2)-d),(a-x(1)),(c-x(2))]);
```

Following the prescription of using the minimum of nonlinear constraint functions, for the L-shaped region, the nonlinear constraint function is:

```
function [c,ceq] = rectconstrfcn(x)

ceq = []; % no equality constraint
F(1) = rectconstr(x,-1,1,0,1); % one rectangle
F(2) = rectconstr(x,0,1,-1,1); % another rectangle
c = min(F); % for OR constraints
```
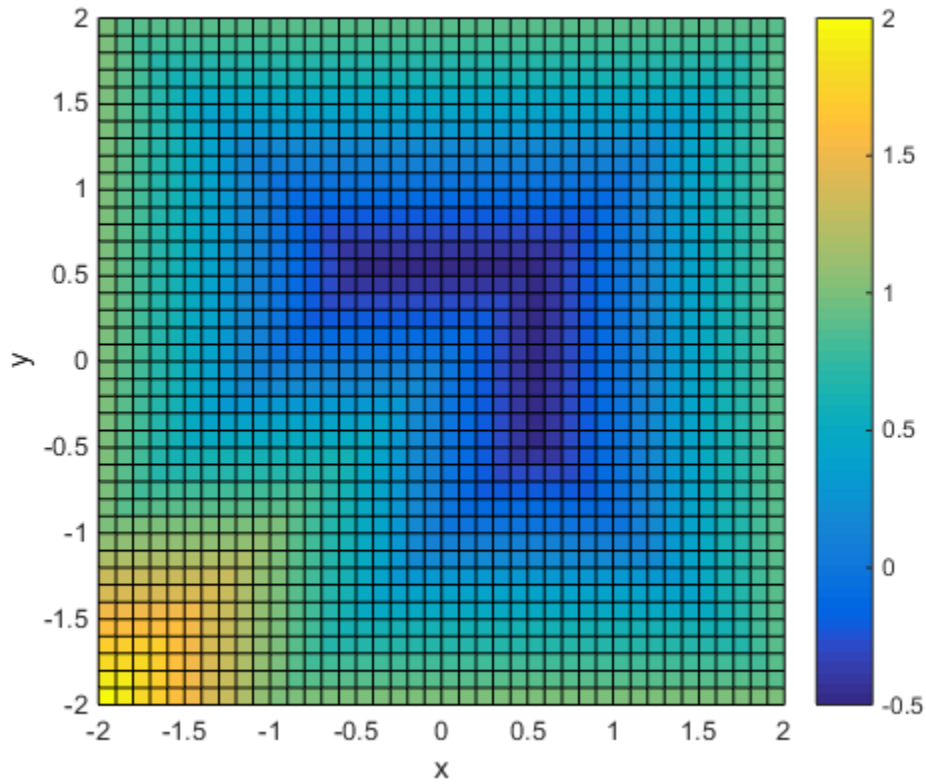
Code for creating the figure

Plot `rectconstrfcn` over the region $\max|x| \le 2$ for $a = -1$, $b = 1$, $c = 0$, $d = 1$:

```matlab
[xx,yy] = meshgrid(-2:.1:2);
x = [xx(:),yy(:)]; % one row per point

z = zeros(length(x),1); % allocate
for ii = 1:length(x)
    [z(ii),~] = rectconstrfcn(x(ii,:));
end

z = reshape(z,size(xx));
surf(xx,yy,z)
```

```
colorbar
axis equal
xlabel('x');ylabel('y')
view(0,90)
```

Suppose your objective function is

```
fun = @(x)exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1);
```

Minimize `fun` over the L-shaped region:

```
opts = optimoptions(@fmincon,'Algorithm','interior-point','Display','off');
x0 = [-.5,.6]; % an arbitrary guess
[xsol,fval,eflag,output] = fmincon(fun,x0,[],[],[],[],[],[],@rectconstrfcn,opts)

xsol =

    0.4998   -0.9996


fval =

   2.4649e-07


eflag =

     1


output =

         iterations: 17
          funcCount: 59
     constrviolation: 0
           stepsize: 1.8763e-04
          algorithm: 'interior-point'
       firstorderopt: 4.9302e-07
        cgiterations: 0
             message: 'Local minimum found that satisfies the constraints.…'
```

Clearly, the solution `xsol` is inside the L-shaped region. The exit flag is `1`, indicating that `xsol` is a local minimum.

How to Use 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 local function `myobj(x)`. The nonlinear constraints are in the local function `myconstr(x)`. This example does not use gradients.

```
function [x fval exitflag] = fullexample
x0 = [1; 4; 5; 2; 5];
lb = [-Inf; -Inf;  0; -Inf;   1];
ub = [ Inf;  Inf; 20; Inf; Inf];
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];
opts = optimoptions(@fmincon,'Algorithm','sqp');

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

%------------------------------------------------------------
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:

```
[x fval exitflag] = fullexample;

Local minimum found that satisfies the constraints.

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

x =
    0.6114
    2.0380
    1.3948
    0.1572
```

```
     1.5498

fval =
   37.3806

exitflag =
     1
```

Objective and Nonlinear Constraints in the Same Function

This example shows how to avoid calling a function twice when it computes values for both objective and constraints.

You typically use such a function in a simulation. Solvers such as `fmincon` evaluate the objective and nonlinear constraint functions separately. This evaluation is wasteful when you use the same calculation for both results.

To avoid wasting time, have your calculation use a nested function to evaluate the objective and constraint functions only when needed, by retaining the values of time-consuming calculations. Using a nested function avoids using global variables, yet lets intermediate results be retained and shared between the objective and constraint functions.

Step 1. Function that computes objective and constraints.

For example, suppose `computeall` is the expensive (time-consuming) function called by both the objective function and by the nonlinear constraint functions. Suppose you want to use `fmincon` as your optimizer.

Write a function that computes a portion of Rosenbrock's function `f1` and a nonlinear constraint `c1` that keeps the solution in a disk of radius 1 around the origin. Rosenbrock's function is

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

which has a unique minimum value of 0 at (1,1). See "Solve a Constrained Nonlinear Problem" on page 1-3.

In this example there is no nonlinear equality constraint, so `ceq1 = []`. Add a `pause(1)` statement to simulate an expensive computation.

```
function [f1,c1,ceq1] = computeall(x)
    ceq1 = [];
    c1 = norm(x)^2 - 1;
    f1 = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;
    pause(1) % simulate expensive computation
end
```

Save `computeall.m` as a file on your MATLAB path.

Step 2. Embed function in nested function that keeps recent values.

Suppose the objective function is
$y = 100(x_2 - x_1{}^2)^2 + (1 - x_1)^2$
$+ 20*(x_3 - x_4{}^2)^2 + 5*(1 - x_4)^2.$

`computeall` returns the first part of the objective function. Embed the call to `computeall` in a nested function:

```matlab
function [x,f,eflag,outpt] = runobjconstr(x0,opts)

if nargin == 1 % No options supplied
    opts = [];
end

xLast = []; % Last place computeall was called
myf = []; % Use for objective at xLast
myc = []; % Use for nonlinear inequality constraint
myceq = []; % Use for nonlinear equality constraint

fun = @objfun; % the objective function, nested below
cfun = @constr; % the constraint function, nested below

% Call fmincon
[x,f,eflag,outpt] = fmincon(fun,x0,[],[],[],[],[],[],cfun,opts);

    function y = objfun(x)
        if ~isequal(x,xLast) % Check if computation is necessary
            [myf,myc,myceq] = computeall(x);
            xLast = x;
        end
        % Now compute objective function
        y = myf + 20*(x(3) - x(4)^2)^2 + 5*(1 - x(4))^2;
    end

    function [c,ceq] = constr(x)
        if ~isequal(x,xLast) % Check if computation is necessary
            [myf,myc,myceq] = computeall(x);
            xLast = x;
        end
        % Now compute constraint functions
        c = myc; % In this case, the computation is trivial
```

```
        ceq = myceq;
    end

end
```

Save the nested function as a file named `runobjconstr.m` on your MATLAB path.

Step 3. Time to run with nested function.

Run the file, timing the call with `tic` and `toc`.

```
opts = optimoptions(@fmincon,'Algorithm','interior-point','Display','off');
x0 = [-1,1,1,2];
tic
[x,fval,exitflag,output] = runobjconstr(x0,opts);
toc
```

```
Elapsed time is 203.797275 seconds.
```

Step 4. Time to run without nested function.

Compare the times to run the solver with and without the nested function. For the run without the nested function, save `myrosen2.m` as the objective function file, and `constr.m` as the constraint:

```
function y = myrosen2(x)
    f1 = computeall(x); % get first part of objective
    y = f1 + 20*(x(3) - x(4)^2)^2 + 5*(1 - x(4))^2;
end

function [c,ceq] = constr(x)
    [~,c,ceq] = computeall(x);
end
```

Run `fmincon`, timing the call with `tic` and `toc`.

```
tic
[x,fval,exitflag,output] = fmincon(@myrosen2,x0,...
                    [],[],[],[],[],[],@constr,opts);
toc
```

```
Elapsed time is 406.771978 seconds.
```

The solver takes twice as long as before, because it evaluates the objective and constraint separately.

Step 5. Save computing time with parallel computing.

If you have a Parallel Computing Toolbox license, you can save even more time by setting the `UseParallel` option to `true`.

```
parpool
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

ans =

 Pool with properties:

             Connected: true
            NumWorkers: 4
               Cluster: local
         AttachedFiles: {}
           IdleTimeout: 30 minute(s) (30 minutes remaining)
           SpmdEnabled: true
```

```
opts = optimoptions(opts,'UseParallel',true);
tic
[x,fval,exitflag,output] = runobjconstr(x0,opts);
toc
```

```
Elapsed time is 97.528110 seconds.
```

In this case, enabling parallel computing cuts the computational time in half.

Compare the runs with parallel computing, with and without a nested function:

```
tic
[x,fval,exitflag,output] = fmincon(@myrosen2,x0,...
                    [],[],[],[],[],[],@constr,opts);
toc
```

```
Elapsed time is 188.985178 seconds.
```

In this example, computing in parallel but not nested takes about the same time as computing nested but not parallel. Computing both nested and parallel takes half the time of using either alone.

Related Examples

- "Solve a Constrained Nonlinear Problem" on page 1-3

More About

- "Optimizing a Simulation or Ordinary Differential Equation" on page 4-31
- "Parallel Computing"

Passing Extra Parameters

Extra Parameters, Fixed Variables, or Data

Sometimes objective or constraint functions have parameters in addition to the independent variable. The extra parameters can be data, or can represent variables that do not change during the optimization. There are three methods of passing these parameters:

- "Anonymous Functions" on page 2-50
- "Nested Functions" on page 2-52
- "Global Variables" on page 2-52

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

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

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 a 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:

```
Local minimum found.

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

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 `f` 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 2-2 via a nested function, write a single 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 function 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
```

The objective function is 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 2-50.

Global Variables

Global variables can be troublesome, so 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 a function file:

```
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 2-50.

What Are Options?

Options are a way of combining a set of name-value pairs. They are useful because they allow you to:

- Tune or modify the optimization process.
- Select extra features, such as output functions and plot functions.
- Save and reuse settings.

They simplify solver syntax—you don't have to include a lot of name-value pairs in a call to a solver.

To see how to set and change options, see "Set and Change Options" on page 2-56.

For an overview of all options, including which solvers use each option, see "Optimization Options Reference" on page 13-7.

Options in Common Use: Tuning and Troubleshooting

You set or change options when the default settings do not work sufficiently well. This can mean the solver takes too long to converge, the solver fails, or you are unsure of the reliability of the result.

To tune your solver for improved speed or accuracy, try setting these options first:

- "Choosing the Algorithm" on page 2-7 — `Algorithm`
- "Tolerances and Stopping Criteria" on page 2-61 — `TolFun`, `TolX`, `MaxFunEvals`, and `MaxIter`
- Finite differences — `FinDiffType` and `FinDiffRelStep`

To diagnose and troubleshoot, try setting these options first:

- "Iterative Display" on page 3-16 — `Display`
- Function evaluation errors — `FunValCheck`
- "Plot Functions" on page 3-29 and "Output Functions" on page 3-35 — `PlotFcns` and `OutputFunction`

See Also
`optimoptions` | `optimset`

Related Examples
- "Improve Results"

More About
- "Solver Outputs and Iterative Display"

Set and Change Options

The recommended way to set options is to use the `optimoptions` function. For example, to set the `fmincon` algorithm to `sqp`, set iterative display, and set a small value of the `TolCon` tolerance:

```
options = optimoptions('fmincon',...
    'Algorithm','sqp','Display','iter','TolCon',1e-12);
```

Note: Use `optimset` instead of `optimoptions` for the `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg` solvers. These are the solvers that do not require an Optimization Toolbox license.

Change options as follows:

- Dot notation. For example,

    ```
    options.TolX = 1e-10;
    ```

- `optimoptions`. For example,

    ```
    options = optimoptions(options,'TolX',1e-10);
    ```

Ensure that you pass `options` in your solver call. For example,

```
[x,fval] = fmincon(@objfun,x0,[],[],[],[],lb,ub,@nonlcon,options);
```

You can also set and change options using the "Optimization App" on page 5-2.

Choose Between optimoptions and optimset

Previously, the recommended way to set options was to use `optimset`. Now the general recommendation is to use `optimoptions`, with some caveats listed below.

`optimset` still works, and it is the only way to set options for solvers that are available without an Optimization Toolbox license: `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg`.

Note: Some other toolboxes use optimization options and require you to pass in options created using `optimset`, not `optimoptions`. Check the documentation for your toolboxes.

`optimoptions` organizes options by solver, with a more focused and comprehensive display than `optimset`:

- Creates and modifies only the options that apply to a solver
- Shows your option choices and default values for a specific solver/algorithm
- Displays links for more information on solver options and other available solver algorithms

`intlinprog` uses only `optimoptions` options.

The main difference in creating options is:

- For `optimoptions`, you include the solver name as the first argument.

  ```
  options = optimoptions(SolverName,Name,Value,...)
  ```
- For `optimset`, the syntax does not include the solver name.

  ```
  options = optimset(Name,Value,...)
  ```

In both cases, you can query or change options by using dot notation. See "Set and Change Options" on page 2-56 and "View Options" on page 2-59.

For example, compare the display of `optimoptions` to that of `optimset`.

```
options = optimset('GradObj','on')
options = optimoptions(@fminunc,'GradObj','on')
```

```
options =                                              options =

  fminunc options:                                                Displ
                                                            MaxFunEva
  Options used by current Algorithm ('trust-region'):           MaxIt
  (Other available algorithms: 'quasi-newton')                   TolF
                                                                    To
  Set by user:                                              FunValChe
   GradObj: 'on'                                              OutputF
                                                              PlotFc
  Default:                                                 ActiveConstrI
          Algorithm: 'trust-region'                           Algorit
    DerivativeCheck: 'off'                           AlwaysHonorConstrain
       Diagnostics: 'off'                                   BranchStrate
     DiffMaxChange: Inf                                     DerivativeChe
     DiffMinChange: 0                                          Diagnosti
           Display: 'final'                                   DiffMaxChan
    FinDiffRelStep: 1.4901e-08                                DiffMinChan
       FinDiffType: 'forward'                               FinDiffRelSt
       FunValCheck: 'off'                                    FinDiffTy
          HessMult: []                                   GoalsExactAchie
       HessPattern: 'sparse(ones(numberOfVariables))'        GradCons
          Hessian: 'off'                                        GradO
       MaxFunEvals: '100*numberOfVariables'                     HessF
           MaxIter: 400                                        Hessi
         MaxPCGIter: 'max(1,floor(numberOfVariables/2))'       HessMu
         OutputFcn: []                                      HessPatte
           PlotFcns: []                                     HessUpda
   PrecondBandWidth: 0                                    InitialHessTy
            TolFun: 1.0000e-06                            InitialHessMatr
            TolPCG: 0.1000                                 InitBarrierPar
              TolX: 1.0000e-06                         InitTrustRegionRadi
          TypicalX: 'ones(numberOfVariables,1)'                  Jacobi
                                                                JacobMu
  Show options not used by current Algorithm ('trust-region')  JacobPatte
                                                               LargeSca
                                                             LineSearchTy
                                                                 MaxNod
                                                                MaxPCGIt
                                                             MaxProjCGIt
                                                                MaxRLPIt
                                                                MaxSQPIt
                                                                  MaxTi
                                                             MeritFuncti
                                                                 MinAbsM
                                                           NodeDisplayInterv
                                                             NodeSearchStrate
```

View Options

You can view the value of a particular option by using dot notation. For example,

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

To view the value of the TolX tolerance:

```
options.TolX

ans =

   1.0000e-10
```

You can view the value of all options, and see which ones have nondefault values, by entering the options name:

```
options
```

```
options =

  fmincon options:

  Options used by current Algorithm ('interior-point'):
  (Other available algorithms: 'active-set', 'sqp', 'trust-region-reflective')

  Set by user:
   Algorithm: 'interior-point'

  Default:
   AlwaysHonorConstraints: 'bounds'
           DerivativeCheck: 'off'
               Diagnostics: 'off'
             DiffMaxChange: Inf
             DiffMinChange: 0
                   Display: 'final'
            FinDiffRelStep: 1.4901e-08
               FinDiffType: 'forward'
               FunValCheck: 'off'
                GradConstr: 'off'
                   GradObj: 'off'
                   HessFcn: []
                  HessMult: []
                   Hessian: 'bfgs'
          InitBarrierParam: 0.1000
     InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                MaxFunEvals: 3000
                    MaxIter: 1000
             MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
            ObjectiveLimit: -1.0000e+20
                 OutputFcn: []
                  PlotFcns: []
              ScaleProblem: 'none'
       SubproblemAlgorithm: 'ldl-factorization'
                    TolCon: 1.0000e-06
                    TolFun: 1.0000e-06
                 TolProjCG: 0.0100
              TolProjCGAbs: 1.0000e-10
                      TolX: 1.0000e-10
                  TypicalX: 'ones(numberOfVariables,1)'
               UseParallel: 'never'
```

Tolerances and Stopping Criteria

The number of iterations in an optimization depends on a solver's *stopping criteria*. These criteria include several *tolerances* you can set. Generally, a tolerance is a threshold which, if crossed, stops the iterations of a solver.

Set tolerances and other criteria using `optimoptions` as explained in "Set and Change Options" on page 2-56.

Tip Generally set tolerances such as `TolFun` and `TolX` to be well above `eps`, and usually above `1e-14`. Setting small tolerances does not always result in accurate results. Instead, a solver can fail to recognize when it has converged, and can continue futile iterations. A tolerance value smaller than `eps` effectively disables that stopping condition.

You can find the default tolerances in the "Optimization App" on page 5-2. Some default tolerances differ for different algorithms, so set both the solver and the algorithm.
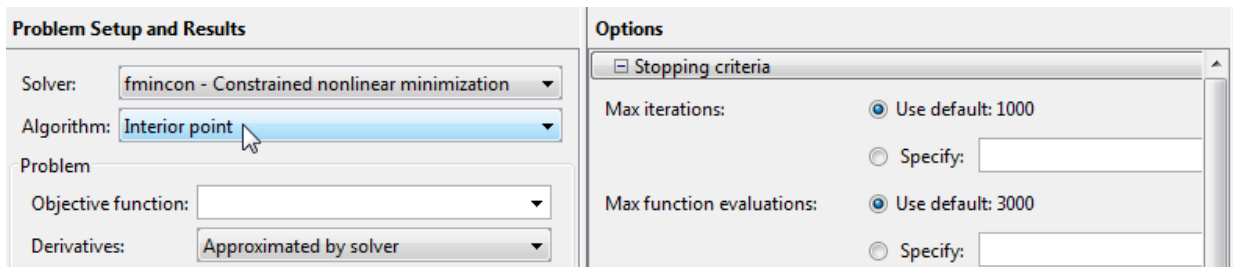
`optimoptions` displays default tolerances. For example,

```
options = optimoptions('fmincon')
```

```
options =

  fmincon options:

  Options used by current Algorithm ('interior-point'):
  (Other available algorithms: 'active-set', 'sqp', 'trust-region-reflective')

  Set by user:
    No options set by user.

  Default:
                  Algorithm: 'interior-point'
    AlwaysHonorConstraints: 'bounds'
            DerivativeCheck: 'off'
                Diagnostics: 'off'
              DiffMaxChange: Inf
              DiffMinChange: 0
                    Display: 'final'
              FinDiffRelStep: 'sqrt(eps)'
                FinDiffType: 'forward'
                FunValCheck: 'off'
                  GradConstr: 'off'
                    GradObj: 'off'
                    HessFcn: []
                    Hessian: 'bfgs'
                    HessMult: []
            InitBarrierParam: 0.1000
      InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                MaxFunEvals: 3000
                    MaxIter: 1000
              MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
              ObjectiveLimit: -1.0000e+20
                  OutputFcn: []
                    PlotFcns: []
              ScaleProblem: 'none'
        SubproblemAlgorithm: 'ldl-factorization'
                    TolCon: 1.0000e-06
                    TolFun: 1.0000e-06
                  TolProjCG: 0.0100
              TolProjCGAbs: 1.0000e-10
                      TolX: 1.0000e-10
                  TypicalX: 'ones(numberOfVariables,1)'
```

You can also find the default tolerances in the options section of the solver function reference page.

- 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.
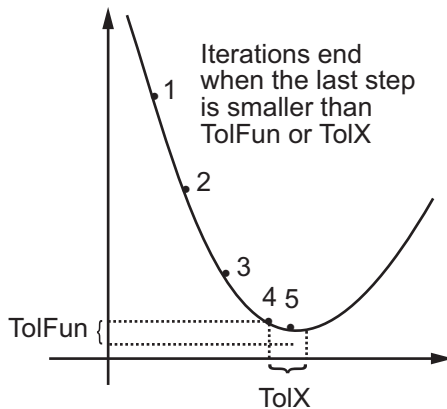
- For some algorithms, TolFun is a lower bound on the change in the value of the objective function during a step. For those algorithms, if $|f(x_i) - f(x_{i+1})| <$ TolFun, the iterations end. TolFun is sometimes used as a *relative* bound, meaning iterations end when $|f(x_i) - f(x_{i+1})| <$ TolFun*$(1 + |f(x_i)|)$, or a similar relative measure.

Note: TolFun is most often 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 on the first-order optimality measure. First-order optimality measure is defined in "First-Order Optimality Measure" on page 3-11.

- 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.

- `MaxIter` is a bound on the number of solver iterations. `MaxFunEvals` is a bound on the number of function evaluations. Iterations and function evaluations are discussed in "Iterations and Function Counts" on page 3-10.

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 6-23.

There are several tolerances that apply only to the `fmincon` interior-point algorithm. For more information, see **Interior-Point Algorithm** in `fmincon` options.

There are several tolerances that apply only to `intlinprog`. See "Some "Integer" Solutions Are Not Integers" on page 8-47 and "Branch and Bound" on page 8-42.

Checking Validity of Gradients or Jacobians

In this section...
"How to Check Derivatives" on page 2-65
"Example: Checking Derivatives of Objective and Constraint Functions" on page 2-66

Many solvers allow you to supply a function that calculates first derivatives (gradients or Jacobians) of objective or constraint functions. You can check whether the derivatives calculated by your function match finite-difference approximations. This check can help you diagnose whether your derivative function is correct.

- If a component of the gradient function is less than 1, "match" means the absolute difference of the gradient function and the finite-difference approximation of that component is less than `1e-6`.

- Otherwise, "match" means that the relative difference is less than `1e-6`.

The `DerivativeCheck` option causes the solver to check the supplied derivative against a finite-difference approximation at just one point. If the finite-difference and supplied derivatives do not match, the solver errors. If the derivatives match to within `1e-6`, the solver reports the calculated differences, and continues iterating without further derivative checks. Solvers check the match at a point that is a small random perturbation of the initial point `x0`, modified to be within any bounds. Solvers do not include the computations for `DerivativeCheck` in the function count; see "Iterations and Function Counts" on page 3-10.

How to Check Derivatives

- At the MATLAB command line:

  1 Set the `GradObj`, `GradConstr`, or `Jacobian` options to `'on'` with `optimoptions`. Make sure your objective or constraint functions supply the appropriate derivatives.

  2 Set the `DerivativeCheck` option to `'on'`.

- Using the Optimization app:

  1 In the **Problem Setup and Results** pane, choose **Derivatives**: **Objective function**: `Gradient supplied` or **Nonlinear constraint function**:

> **Derivatives**: `Gradient supplied`. Make sure your objective or constraint functions supply the appropriate derivatives.

2 In the **Options** pane, check **User-supplied derivatives > Validate user-supplied derivatives**

Central finite differences are more accurate than the default forward finite differences. To use central finite differences:

- At the MATLAB command line, set `FinDiffType` option to `'central'` with `optimoptions`.
- Using the Optimization app, in the **Approximated derivatives** pane, set **Type** to `central differences`.

Example: Checking Derivatives of Objective and Constraint Functions

- "Objective and Constraint Functions" on page 2-66
- "Checking Derivatives at the Command Line" on page 2-67
- "Checking Derivatives with the Optimization App" on page 2-68

Objective and Constraint Functions

Consider the problem of minimizing the Rosenbrock function within the unit disk as described in "Solve a Constrained Nonlinear Problem" on page 1-3. The `rosenboth` function calculates the objective function and its gradient:

```
function [f g H] = rosenboth(x)

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

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

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

`rosenboth` calculates the Hessian, too, but this example does not use the Hessian.

The `unitdisk2` function correctly calculates the constraint function and its gradient:

```
function [c,ceq,gc,gceq] = unitdisk2(x)
c = x(1)^2 + x(2)^2 - 1;
ceq = [ ];

if nargout > 2
    gc = [2*x(1);2*x(2)];
    gceq = [];
end
```

The `unitdiskb` function incorrectly calculates gradient of the constraint function:

```
function [c ceq gc gceq] = unitdiskb(x)
c = x(1)^2 + x(2)^2 - 1;
ceq = [ ];

if nargout > 2
    gc = [x(1);x(2)]; % Gradient incorrect: off by a factor of 2
    gceq = [];
end
```

Checking Derivatives at the Command Line

1 Set the options to use the interior-point algorithm, gradient of objective and constraint functions, and the `DerivativeCheck` option:

```
% For reproducibility--DerivativeCheck randomly perturbs the initial point
rng(0,'twister');
options = optimoptions(@fmincon,'Algorithm','interior-point',...
    'DerivativeCheck','on','GradObj','on','GradConstr','on');
```

2 Solve the minimization with `fmincon` using the erroneous `unitdiskb` constraint function:

```
[x fval exitflag output] = fmincon(@rosenboth,...
    [-1;2],[],[],[],[],[],[],@unitdiskb,options);

_____
    Derivative Check Information

Objective function derivatives:
Maximum relative difference between user-supplied
and finite-difference derivatives = 1.84768e-008.

Nonlinear inequality constraint derivatives:
Maximum relative difference between user-supplied
and finite-difference derivatives = 1.
 User-supplied constraint derivative element (2,1):    1.99838
```

```
 Finite-difference constraint derivative element (2,1): 3.99675
_____


Error using validateFirstDerivatives
Derivative Check failed:
User-supplied and forward finite-difference derivatives
do not match within 1e-006 relative tolerance.

Error in fmincon at 805
    validateFirstDerivatives(funfcn,confcn,X, ...
```

The constraint function does not match the calculated gradient, encouraging you to check the function for an error.

3 Replace the `unitdiskb` constraint function with `unitdisk2` and run the minimization again:

```
[x fval exitflag output] = fmincon(@rosenboth,...
   [-1;2],[],[],[],[],[],[],@unitdisk2,options);
```

```
_____

   Derivative Check Information

Objective function derivatives:
Maximum relative difference between user-supplied
and finite-difference derivatives = 1.28553e-008.

Nonlinear inequality constraint derivatives:
Maximum relative difference between user-supplied
and finite-difference derivatives = 1.46443e-008.

Derivative Check successfully passed.
_____


Local minimum found that satisfies the constraints...
```

Checking Derivatives with the Optimization App

Note: The Optimization app warns that it will be removed in a future release.

To set up the example using correct derivative functions, but starting from [0 0], using the Optimization app:

1 Launch the Optimization app by entering `optimtool` at the command line.
2 Set the **Problem Setup and Results** pane to match the following figure:

Problem Setup and Results

Solver:	fmincon - Constrained nonlinear minimization
Algorithm:	Interior point

Problem

Objective function:	@rosenboth
Derivatives:	Gradient supplied
Start point:	[0 0]

Constraints:

Linear inequalities:	A:		b:	
Linear equalities:	Aeq:		beq:	
Bounds:	Lower:		Upper:	

Nonlinear constraint function: @unitdisk2
Derivatives: Gradient supplied

3 Set the **Options** pane to match the following figure:

4 Press the **Start** button under **Run solver and view results**.

The output screen displays
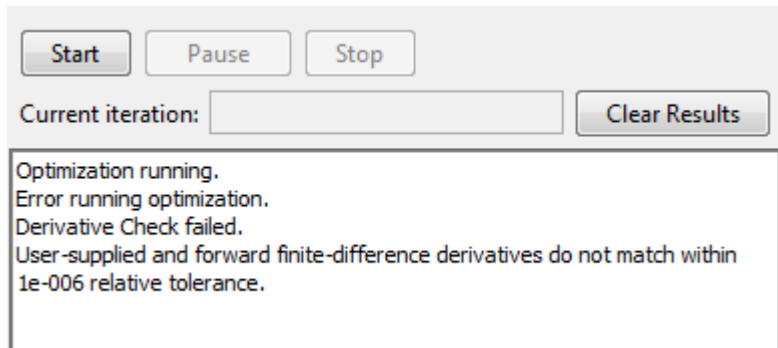
The forward finite difference approximation is inaccurate enough near `[0 0]` that the derivative check fails.

5 To use the more accurate central differences, select `central differences` in the **Approximated derivatives > Type** pane:
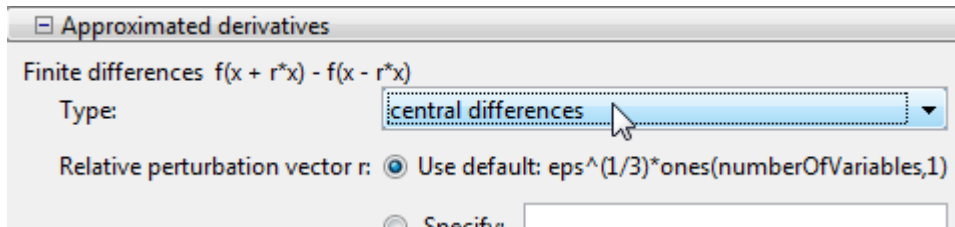
6 Click **Run solver and view results > Clear Results**, then **Start**. This time the derivative check is successful:

```
 Start      Pause      Stop

Current iteration: 24                    Clear Results

----------------------------
Optimization running.
Objective function value: 0.0456748247578765
Local minimum found that satisfies the constraints.

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

The derivative check also succeeds when you select the initial point [-1 2], or most random points.

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.B., A. Orden, and P. Wolfe, "Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints," *Pacific Journal Math.,* Vol. 5, pp. 183–195, 1955.

[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*, Second Edition. Springer Series in Operations Research, Springer Verlag, 2006.

[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*, 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," *Computational Optimization and Applications*, Vol. 20, No. 1, pp. 61–72, 2001.

[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.

[58] Dolan, Elizabeth D. , Jorge J. Moré and Todd S. Munson, "Benchmarking Optimization Software with COPS 3.0," Argonne National Laboratory Technical Report ANL/MCS-TM-273, February 2004.

[59] Applegate, D. L., R. E. Bixby, V. Chvátal and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, 2007.

[60] Spellucci, P., "A new technique for inconsistent QP problems in the SQP method," *Journal of Mathematical Methods of Operations Research*, Volume 47, Number 3, pp. 355–400, October 1998.

[61] Tone, K., "Revisions of constraint approximations in the successive QP method for nonlinear programming problems," *Journal of Mathematical Programming*, Volume 26, Number 2, pp. 144–152, June 1983.

[62] Gondzio, J. "Multiple centrality corrections in a primal dual method for linear programming." *Computational Optimization and Applications*, Volume 6, Number 2, pp. 137–156, 1996.

[63] Gould, N. and P. L. Toint. "Preprocessing for quadratic programming." *Math. Programming*, Series B, Vol. 100, pp. 95–132, 2004.

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

3

Examining Results

Current Point and Function Value

The *current point* and *function value* are the first two outputs of all Optimization Toolbox solvers.

- The current point is the final point in the solver iterations. It is the best point the solver found in its run.

  - If you call a solver without assigning a value to the output, the default output, ans, is the current point.

- The function value is the value of the objective function at the current point.

  - The function value for least-squares solvers is the sum of squares, also known as the residual norm.

  - fgoalattain, fminimax, and fsolve return a vector function value.

  - Sometimes fval or Fval denotes function value.

Exit Flags and Exit Messages

In this section...
"Exit Flags" on page 3-3
"Exit Messages" on page 3-4
"Enhanced Exit Messages" on page 3-5
"Exit Message Options" on page 3-8

Exit Flags

When an optimization solver completes its task, it sets an *exit flag*. An exit flag is an integer that is a code for the reason the solver halted its iterations. In general:

- Positive exit flags correspond to successful outcomes.
- Negative exit flags correspond to unsuccessful outcomes.
- The zero exit flag corresponds to the solver being halted by exceeding an iteration limit or limit on the number of function evaluations (see "Iterations and Function Counts" on page 3-10, and also see "Tolerances and Stopping Criteria" on page 2-61).

A table of solver outputs in the solver's function reference section lists the meaning of each solver's exit flags.

Note: Exit flags are not infallible guides to the quality of a solution. Many other factors, such as tolerance settings, can affect whether a solution is satisfactory to you. You are responsible for deciding whether a solver returns a satisfactory answer. Sometimes a negative exit flag does not correspond to a "bad" solution. Similarly, sometimes a positive exit flag does not correspond to a "good" solution.

You obtain an exit flag by calling a solver with the `exitflag` syntax. This syntax depends on the solver. For details, see the solver function reference pages. For example, for `fsolve`, the calling syntax to obtain an exit flag is

```
[x,fval,exitflag] = fsolve(...)
```

The following example uses this syntax. Suppose you want to solve the system of nonlinear equations

$$2x_1 - x_2 = e^{-x_1}$$

$$-x_1 + 2x_2 = e^{-x_2}.$$

Write these equations as an anonymous function that gives a zero vector at a solution:

```
myfcn = @(x)[2*x(1) - x(2) - exp(-x(1));
      -x(1) + 2*x(2) - exp(-x(2))];
```

Call `fsolve` with the `exitflag` syntax at the initial point [-5 -5]:

```
[xfinal fval exitflag] = fsolve(myfcn,[-5 -5])

Equation solved.

fsolve completed because the vector of function values is near
zero as measured by the default value of the function tolerance,
and the problem appears regular as measured by the gradient.

xfinal =
    0.5671    0.5671

fval =
  1.0e-006 *
   -0.4059
   -0.4059

exitflag =
     1
```

In the table for `fsolve` exitflag, you find that an exit flag value `1` means "Function converged to a solution x." In other words, `fsolve` reports `myfcn` is nearly zero at `x = [0.5671 0.5671]`.

Exit Messages

Each solver issues a message to the MATLAB command window at the end of its iterations. This message explains briefly why the solver halted. The message might give more detail than the exit flag.

Many examples in this documentation show exit messages. For example, see "Minimizing at the Command Line" on page 1-10, or "Step 3: Invoke fminunc using the options." on

page 6-13. The example in the previous section, "Exit Flags" on page 3-3, shows the following exit message:

```
Equation solved.

fsolve completed because the vector of function values is near
zero as measured by the default value of the function tolerance,
and the problem appears regular as measured by the gradient.
```

This message is more informative than the exit flag. The message indicates that the gradient is relevant. The message also states that the function tolerance controls how near 0 the vector of function values must be for fsolve to regard the solution as completed.

Enhanced Exit Messages

Some solvers have exit messages that contain links for more information. There are two types of links:

- Links on words or phrases. If you click such a link, a window opens that displays a definition of the term, or gives other information. The new window can contain links to the Help browser documentation for more detailed information.
- A link as the last line of the display saying <stopping criteria details>. If you click this link, MATLAB displays more detail about the reason the solver halted.

The fminunc solver has enhanced exit messages:

```
opts = optimoptions(@fminunc,'Algorithm','quasi-newton'); % 'trust-region' needs gradi
[xfinal fval exitflag] = fminunc(@sin,0,opts)
```

This yields the following results:

```
Local minimum found.

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

<stopping criteria details>


xfinal =

   -1.5708


fval =

   -1.0000


exitflag =

      1
```

Each of the underlined words or phrases contains a link that provides more information.

- The `<stopping criteria details>` link prints the following to the MATLAB command line:

  ```
  Optimization completed: The first-order optimality measure, 0.000000e+000, is less
  than the default value of options.TolFun = 1.000000e-006.

  Optimization Metric                                User Options
  relative norm(gradient) =  0.00e+000              TolFun =  1e-006 (default)
  ```

- The other links bring up a help window with term definitions. For example, clicking the `Local minimum found` link opens the following window:
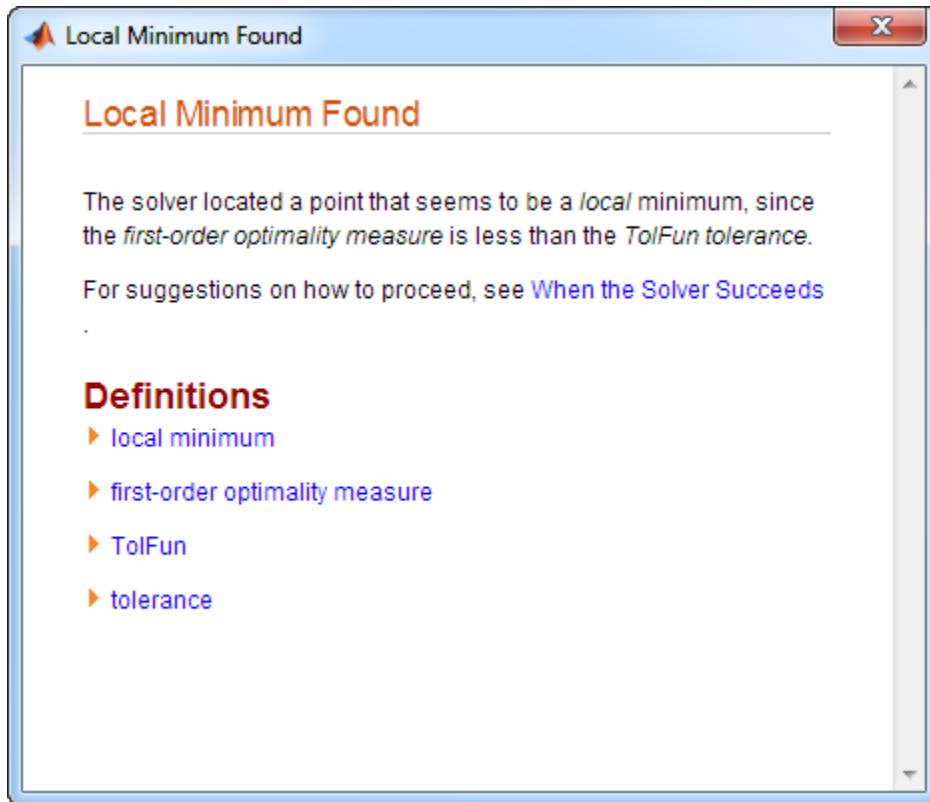
Clicking the `first-order optimality measure` expander link brings up the definition of first-order optimality measure for `fminunc`:
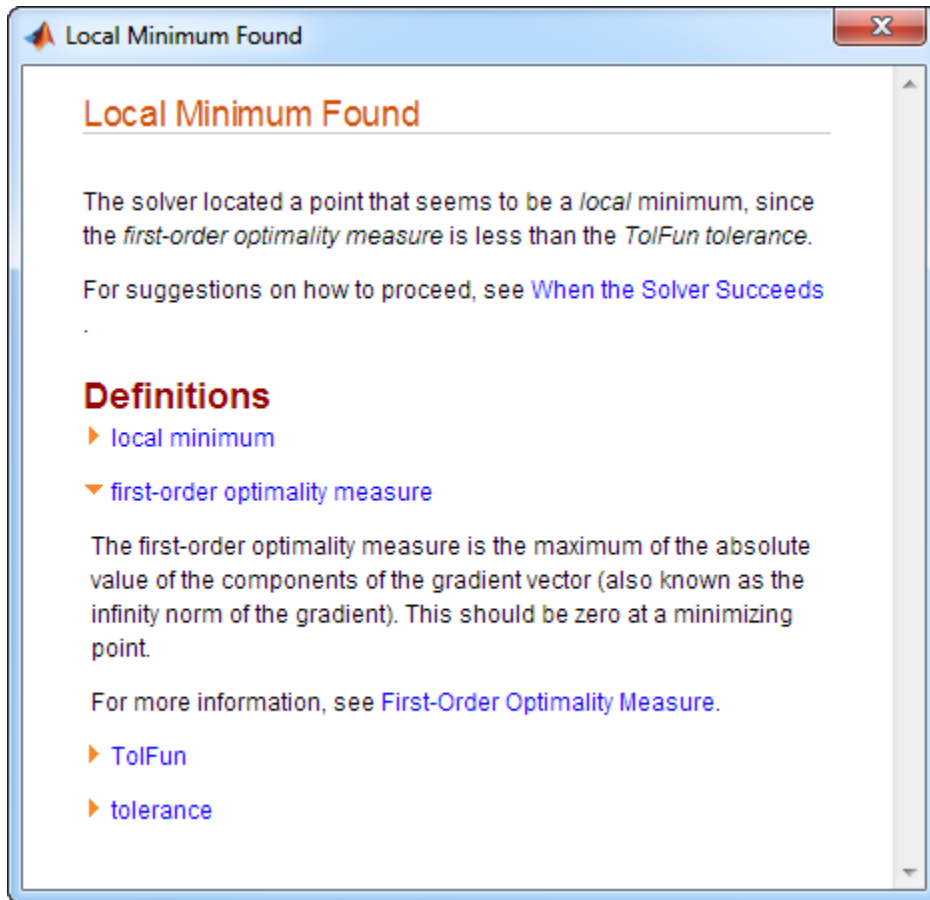
The expander link is a way to obtain more information in the same window. Clicking the first-order optimality measure expander link again closes the definition.

- The other links open the Help Viewer.

Exit Message Options

Set the Display option to control the appearance of both exit messages and iterative display. For more information, see "Iterative Display" on page 3-16. The following table shows the effect of the various settings of the Display option.

Value of the Display Option	Output to Command Window	
	Exit message	Iterative Display
`'none'`, or the synonymous `'off'`	None	None
`'final'` (default for most solvers)	Default	None
`'final-detailed'`	Detailed	None
`'iter'`	Default	Yes
`'iter-detailed'`	Detailed	Yes
`'notify'`	Default only if $\text{exitflag} \leq 0$	None
`'notify-detailed'`	Detailed only if $\text{exitflag} \leq 0$	None

For example,

```
opts = optimoptions(@fminunc,'Display','iter-detailed','Algorithm','quasi-newton');
[xfinal fval] = fminunc(@cos,1,opts);
```

yields the following display:

```
>> opts = optimoptions(@fminunc,'Display','iter-detailed','Algorithm','quasi-newton');
[xfinal fval] = fminunc(@cos,1,opts);
                                                   First-order
 Iteration   Func-count        f(x)      Step-size    optimality
      0           2         0.540302                    0.841
      1           6        -0.990628      2.38223       0.137
      2          10              -1      0.351894      0.000328
      3          12              -1             1      1.03e-06

Optimization completed: The first-order optimality measure, 5.602276e-07, is less
than options.TolFun = 1.000000e-06.

Optimization Metric                                Options
relative norm(gradient) =    5.60e-07              TolFun =   1e-06 (default)
```

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 1-10.

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 3-24.

Sometimes a solver attempts a step, and rejects the attempt. The `trust-region`, `trust-region-reflective`, and `trust-region-dogleg` algorithms count these failed attempts as iterations, and report the (unchanged) result in the iterative display. The `interior-point`, `active-set`, and `levenberg-marquardt` algorithms do not count such an attempt as an iteration, and do not report the attempt in the iterative display. All attempted steps increase the `F-count`, regardless of the algorithm.

First-Order Optimality Measure

In this section...

What Is First-Order Optimality Measure?

First-order optimality is a measure of how close a point x is to optimal. Most Optimization Toolbox solvers use this measure, though it has different definitions for different algorithms. First-order optimality is a necessary condition, but it is not a sufficient condition. In other words:

- The first-order optimality measure must be zero at a minimum.
- A point with first-order optimality equal to zero is not necessarily a minimum.

For general information about first-order optimality, see Nocedal and Wright [31]. For specifics about the first-order optimality measures for Optimization Toolbox solvers, see "Unconstrained Optimality" on page 3-12, "Constrained Optimality Theory" on page 3-12, and "Constrained Optimality in Solver Form" on page 3-14.

Stopping Rules Related to First-Order Optimality

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

Some solvers or algorithms use *relative* first-order optimality as a stopping criterion. Solver iterations end if the first-order optimality measure is less than μ times `TolFun`, where μ is either:

- The infinity norm (maximum) of the gradient of the objective function at `x0`
- The infinity norm (maximum) of inputs to the solver, such as `f` or `b` in `linprog` or `H` in `quadprog`

A relative measure attempts to account for the scale of a problem. Multiplying an objective function by a very large or small number does not change the stopping condition for a relative stopping criterion, but does change it for an unscaled one.

Solvers with enhanced exit messages state, in the stopping criteria details, when they use relative first-order optimality.

Unconstrained Optimality

For a smooth unconstrained problem,

$$\min_x f(x),$$

the first-order optimality measure is the infinity norm (meaning maximum absolute value) of $\nabla f(x)$, which is:

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

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 near a minimum. If the first-order optimality measure is not small, the objective function is not minimal.

Constrained Optimality Theory

This section summarizes 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 3-14.

For a smooth constrained problem, let g and h be vector functions representing all inequality and equality constraints respectively (meaning 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 complex 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 use the auxiliary Lagrangian function:

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

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

The KKT conditions are:

$$\nabla_x L(x, \lambda) = 0,$$

$$\lambda_{g,i} g_i(x) = 0 \ \forall i,$$

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

Solvers do not use the three expressions in Equation 3-4 in the calculation of optimality measure.

The optimality measure associated with Equation 3-2 is

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

The optimality measure associated with Equation 3-3 is

$$\left\| \overrightarrow{\lambda_g g}(x) \right\|,$$

where the norm in Equation 3-6 means infinity norm (maximum) of the vector $\overrightarrow{\lambda_{g,i} g_i}(x)$.

The combined optimality measure is the maximum of the values calculated in Equation 3-5 and Equation 3-6. Solvers that accept nonlinear constraint functions report constraint violations $g(x) > 0$ or $|h(x)| > 0$ as `TolCon` tolerance violations. See "Tolerances and Stopping Criteria" on page 2-61.

Constrained Optimality in Solver Form

Most constrained toolbox solvers separate their calculation of first-order optimality measure into bounds, linear functions, and nonlinear functions. The measure is the maximum of the following two norms, which correspond to Equation 3-5 and Equation 3-6:

$$\left\| \nabla_x L(x, \lambda) \right\| = \left\| \nabla f(x) + A^T \lambda_{ineqlin} + Aeq^T \lambda_{eqlin} \right.$$
$$\left. + \sum \lambda_{ineqnonlin,i} \nabla c_i(x) + \sum \lambda_{eqnonlin,i} \nabla ceq_i(x) \right\|,$$

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

where the norm of the vectors in Equation 3-7 and Equation 3-8 is the infinity norm (maximum). The subscripts on the Lagrange multipliers correspond to solver Lagrange multiplier structures. See "Lagrange Multiplier Structures" on page 3-25. The summations in Equation 3-7 range over all constraints. If a bound is ±`Inf`, that term is not constrained, so it is not part of the summation.

Linear Equalities Only

For some large-scale problems with only linear equalities, the first-order optimality measure is the infinity norm of the *projected* gradient. In other words, the first-order optimality measure is the size of the gradient projected onto the null space of `Aeq`.

Bounded Least-Squares and Trust-Region-Reflective Solvers

For least-squares solvers and trust-region-reflective algorithms, in problems with bounds alone, the first-order optimality measure is the maximum over i of $|v_i * g_i|$. Here g_i is the ith component of the gradient, x is the current point, and

$$v_i = \begin{cases} |x_i - b_i| & \text{if the negative gradient points toward bound } b_i \\ 1 & \text{otherwise.} \end{cases}$$

If x_i is at a bound, v_i is zero. If x_i is not at a bound, then at a minimizing point the gradient g_i should be zero. Therefore the first-order optimality measure should be zero at a minimizing point.

Iterative Display

In this section...
"Introduction" on page 3-16
"Common Headings" on page 3-17
"Function-Specific Headings" on page 3-17

Introduction

Iterative display is a table of statistics describing the calculations in each iteration of a solver. The statistics depend on both the solver and the solver algorithm. For more information about iterations, see "Iterations and Function Counts" on page 3-10. The table appears in the MATLAB Command Window when you run solvers with appropriate options.

Obtain iterative display by using `optimoptions` to create options with the `Display` option set to `'iter'` or `'iter-detailed'`. For example:

```
options = optimoptions(@fminunc,'Display','iter','Algorithm','quasi-newton');
[x fval exitflag output] = fminunc(@sin,0,options);
```

```
                                        First-order
Iteration  Func-count      f(x)       Step-size      optimality
    0           2            0                              1
    1           4       -0.841471           1            0.54
    2           8          -1        0.484797        0.000993
    3          10          -1           1           5.62e-005
    4          12          -1           1               0
```

```
Local minimum found.

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

You can also obtain iterative display by using the Optimization app. Select **Display to command window** > **Level of display** > `iterative` or `iterative with detailed message`.
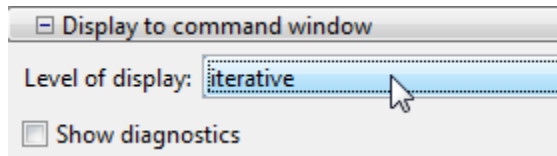
Iterative display is available for all solvers except:

- linprog 'active-set' algorithm
- lsqlin 'trust-region-reflective' and 'active-set' algorithms
- lsqnonneg
- quadprog 'trust-region-reflective' and 'active-set' algorithms

Common Headings

The following table lists some common headings of iterative display.

Heading	Information Displayed
f(x)	Current objective function value. For fsolve, the square of the norm of the function value vector.
First-order optimality	First-order optimality measure (see "First-Order Optimality Measure" on page 3-11).
Func-count or F-count	Number of function evaluations; see "Iterations and Function Counts" on page 3-10.
Iteration or Iter	Iteration number; see "Iterations and Function Counts" on page 3-10.
Norm of step	Size of the current step (size is the Euclidean norm, or 2-norm).

Function-Specific Headings

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

- "fgoalattain, fmincon, fminimax, and fseminf" on page 3-18
- "fminbnd and fzero" on page 3-19
- "fminsearch" on page 3-20

fgoalattain, fmincon, fminimax, and fseminf

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

fgoalattain, fmincon, fminimax, or fseminf Heading	Information Displayed
Attainment factor	Value of the attainment factor for fgoalattain.
CG-iterations	Number of conjugate gradient iterations taken in the current iteration (see "Preconditioned Conjugate Gradient Method" on page 6-23).
Directional derivative	Gradient of the objective function along the search direction.
Feasibility	Maximum constraint violation, where satisfied inequality constraints count as 0.
Line search steplength	Multiplicative factor that scales the search direction (see Equation 6-45).
Max constraint	Maximum violation among all constraints, both internally constructed and user-provided; can be negative when no constraint is binding.
Objective value	Objective function value of the nonlinear programming reformulation of the minimax problem for fminimax.
Procedure	Hessian update procedures: • Infeasible start point • Hessian not updated • Hessian modified

fgoalattain, fmincon, fminimax, or fseminf Heading	Information Displayed
	• `Hessian modified twice` For more information, see "Updating the Hessian Matrix" on page 6-30. QP subproblem procedures: • `dependent` — There are dependent (redundant) equality constraints that the solver detected and removed. • `Infeasible` — The QP subproblem with linearized constraints is infeasible. • `Overly constrained` — The QP subproblem with linearized constraints is infeasible. • `Unbounded` — The QP subproblem is feasible with large negative curvature. • `Ill-posed` — The QP subproblem search direction is too small. • `Unreliable` — The QP subproblem seems to be ill-conditioned.
`Steplength`	Multiplicative factor that scales the search direction (see Equation 6-45).
`Trust-region radius`	Current trust-region radius.

fminbnd and fzero

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

fminbnd or fzero Heading	Information Displayed
`Procedure`	Procedures for `fminbnd`: • `initial` • `golden` (golden section search) • `parabolic` (parabolic interpolation)

fminbnd or fzero Heading	Information Displayed
	Procedures for `fzero`: • `initial` (initial point) • `search` (search for an interval containing a zero) • `bisection` • `interpolation` (linear interpolation or inverse quadratic interpolation)
`x`	Current point for the algorithm

fminsearch

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

fminsearch Heading	Information Displayed
`min f(x)`	Minimum function value in the current simplex.
`Procedure`	Simplex procedure at the current iteration. Procedures include: • `initial simplex` • `expand` • `reflect` • `shrink` • `contract inside` • `contract outside` For details, see "fminsearch Algorithm" on page 6-10.

fminunc

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

fminunc Heading	Information Displayed
`CG-iterations`	Number of conjugate gradient iterations taken in the current iteration (see "Preconditioned Conjugate Gradient Method" on page 6-23)

fminunc Heading	Information Displayed
`Line search steplength`	Multiplicative factor that scales the search direction (see Equation 6-11)

The `fminunc` `'quasi-newton'` algorithm can issue a `skipped update` message to the right of the `First-order optimality` column. This message means that `fminunc` did not update its Hessian estimate, because the resulting matrix would not have been positive definite. The message usually indicates that the objective function is not smooth at the current point.

fsolve

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

fsolve Heading	Information Displayed
`Directional derivative`	Gradient of the function along the search direction
`Lambda`	λ_k value defined in "Levenberg-Marquardt Method" on page 10-7
`Residual`	Residual (sum of squares) of the function
`Trust-region radius`	Current trust-region radius (change in the norm of the trust-region radius)

intlinprog

The following table describes the headings specific to `intlinprog`.

linprog Heading	Information Displayed
`nodes explored`	Cumulative number of explored nodes.
`total time (s)`	Time in seconds since `intlinprog` started.
`num int solution`	Number of integer feasible points found.
`integer fval`	Objective function value of the best integer feasible point found. This is an upper bound for the final objective function value.
`relative gap (%)`	$\dfrac{100(b-a)}{|b|+1},$

linprog Heading	Information Displayed
	where • b is the objective function value of the best integer feasible point. • a is the best lower bound on the objective function value.

linprog

The following table describes the headings specific to linprog.

linprog Heading	Information Displayed
Dual Infeas A'*y+z-w-f	Dual infeasibility.
Duality Gap x'*z+s'*w	Duality gap (see "Interior-Point-Legacy Linear Programming" on page 8-8) between the primal objective and the dual objective. s and w appear only in this equation if there are finite upper bounds.
Objective f'*x	Current objective value.
Primal Infeas A*x-b	Primal infeasibility.
Total Rel Error	Total relative error, described at the end of "Main Algorithm" on page 8-8.

lsqnonlin and lsqcurvefit

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

lsqnonlin or lsqcurvefit Heading	Information Displayed
Directional derivative	Gradient of the function along the search direction
Lambda	λ_k value defined in "Levenberg-Marquardt Method" on page 10-7
Resnorm	Value of the squared 2-norm of the residual at x
Residual	Residual vector of the function

quadprog

The following table describes the headings specific to `quadprog`.

quadprog Heading	Information Displayed
`Feasibility`	Maximum constraint violation, where satisfied inequality constraints count as 0.
`Total relative error`	Total relative error is a measure of infeasibility, as defined in "Total Relative Error" on page 9-6

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 app. All results exported from Optimization app 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 app 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.

Lagrange Multiplier Structures

Constrained optimization involves a set of Lagrange multipliers, as described in "First-Order Optimality Measure" on page 3-11. 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."

Hessian

fminunc Hessian

The Hessian for an unconstrained problem is the matrix of second derivatives of the objective function f:

$$\text{Hessian } H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

- **Quasi-Newton Algorithm** — `fminunc` returns an estimated Hessian matrix at the solution. It computes the estimate by finite differences.

- **Trust-Region Algorithm** — `fminunc` returns a Hessian matrix at the next-to-last iterate.

  - If you supply a Hessian in the objective function, `fminunc` returns this Hessian.

  - If you supply a `HessMult` function, `fminunc` returns the `Hinfo` matrix from the `HessMult` function. For more information, see `HessMult` in the `trust-region` section of the `fminunc` options table.

  - Otherwise, `fminunc` returns an approximation from a sparse finite difference algorithm on the gradients.

This Hessian is accurate for the next-to-last iterate. However, the next-to-last iterate might not be close to the final point.

The reason the `trust-region` algorithm returns the Hessian at the next-to-last point is for efficiency. `fminunc` uses the Hessian internally to compute its next step. When `fminunc` reaches a stopping condition, it does not need to compute the next step, so does not compute the Hessian.

fmincon Hessian

The Hessian for a constrained problem is the Hessian of the Lagrangian. For an objective function *f*, nonlinear inequality constraint vector *c*, and nonlinear equality constraint vector *ceq*, the Lagrangian is

$$L = f + \sum_i \lambda_i c_i + \sum_j \lambda_j ceq_j.$$

The λ_i are Lagrange multipliers; see "First-Order Optimality Measure" on page 3-11 and "Lagrange Multiplier Structures" on page 3-25. The Hessian of the Lagrangian is

$$H = \nabla^2 L = \nabla^2 f + \sum_i \lambda_i \nabla^2 c_i + \sum_j \lambda_j \nabla^2 ceq_j.$$

fmincon has four algorithms, with several options for Hessians, as described in "fmincon Trust Region Reflective Algorithm" on page 6-21, "fmincon Active Set Algorithm" on page 6-26, and "fmincon Interior Point Algorithm" on page 6-37. fmincon returns the following for the Hessian:

*   **active-set or sqp Algorithm** — fmincon returns the Hessian approximation it computes at the next-to-last iterate. fmincon computes a quasi-Newton approximation of the Hessian matrix at the solution in the course of its iterations. This approximation does not, in general, match the true Hessian in every component, but only in certain subspaces. Therefore the Hessian that fmincon returns can be inaccurate. For more details of the active-set calculation, see "SQP Implementation" on page 6-29.

*   **trust-region-reflective Algorithm** — fmincon returns the Hessian it computes at the next-to-last iterate.

    *   If you supply a Hessian in the objective function, fmincon returns this Hessian.
    *   If you supply a HessMult function, fmincon returns the Hinfo matrix from the HessMult function. For more information, see **Trust-Region-Reflective Algorithm** in fmincon options.
    *   Otherwise, fmincon returns an approximation from a sparse finite difference algorithm on the gradients.

    This Hessian is accurate for the next-to-last iterate. However, the next-to-last iterate might not be close to the final point.

The reason the `trust-region-reflective` algorithm returns the Hessian at the next-to-last point is for efficiency. `fmincon` uses the Hessian internally to compute its next step. When `fmincon` reaches a stopping condition, it does not need to compute the next step, so does not compute the Hessian.

- **`interior-point` Algorithm**

  - If the `Hessian` option is `lbfgs` or `fin-diff-grads`, or if you supply a Hessian multiply function (`HessMult`), `fmincon` returns `[]` for the Hessian.

  - If the `Hessian` option is `bfgs` (the default), `fmincon` returns a quasi-Newton approximation to the Hessian at the final point. This Hessian can be inaccurate, as in the `active-set` or `sqp` algorithm Hessian.

  - If the `Hessian` option is `user-supplied`, `fmincon` returns the user-supplied Hessian at the final point.

More About

- "Including Derivatives" on page 2-20
- "Hessian" on page 14-60

Plot Functions

In this section...

Plot an Optimization During Execution

You can plot various measures of progress during the execution of a solver. Set the `PlotFcns` name-value pair in `optimoptions`, and specify one or more plotting functions for the solver to call at each iteration. Pass a function handle or cell array of function handles.

There are a variety of predefined plot functions available. See:

- The `PlotFcns` option description in the solver function reference page
- **Optimization app > Options > Plot functions**

You can also use a custom-written plot function. Write a function file using the same structure as an output function. For more information on this structure, see "Output Function" on page 13-20.

Using a Plot Function

This example shows how to use a plot function to view the progress of the `fmincon` interior-point algorithm. The problem is taken from the Getting Started "Solve a Constrained Nonlinear Problem" on page 1-3. The first part of the example shows how to run the optimization using the Optimization app. The second part shows how to run the optimization from the command line.

Note: The Optimization app warns that it will be removed in a future release.

Running the Optimization Using the Optimization App

1 Write the nonlinear objective and constraint functions, including the derivatives:

```
function [f g H] = rosenboth(x)
% ROSENBOTH returns both the value y of Rosenbrock's function
% and also the value g of its gradient and H the Hessian.
```

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

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

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

Save this file as rosenboth.m.

```
function [c,ceq,gc,gceq] = unitdisk2(x)
% UNITDISK2 returns the value of the constraint
% function for the disk of radius 1 centered at
% [0 0]. It also returns the gradient.

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

if nargout > 2
    gc = [2*x(1);2*x(2)];
    gceq = [];
end
```

Save this file as unitdisk2.m.

2 Start the Optimization app by entering optimtool at the command line.

3 Set up the optimization:

- Choose the fmincon solver.

- Choose the Interior point algorithm.

- Set the objective function to @rosenboth.

- Choose Gradient supplied for the objective function derivative.

- Set the start point to [0 0].

- Set the nonlinear constraint function to @unitdisk2.

- Choose Gradient supplied for the nonlinear constraint derivatives.

Your **Problem Setup and Results** panel should match the following figure.

4 Choose three plot functions in the **Options** pane: **Current point**, **Function value**, and **First order optimality**.

5 Click the **Start** button under **Run solver and view results**.

6 The output appears as follows in the Optimization app.

Run solver and view results

| Start | Pause | Stop |

Current iteration: 24   Clear Results

Optimization running.
Objective function value: 0.0456748247578765
Local minimum found that satisfies the constraints.

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

Final point:

1 ▲	2
0.786	0.618

In addition, the following three plots appear in a separate window.
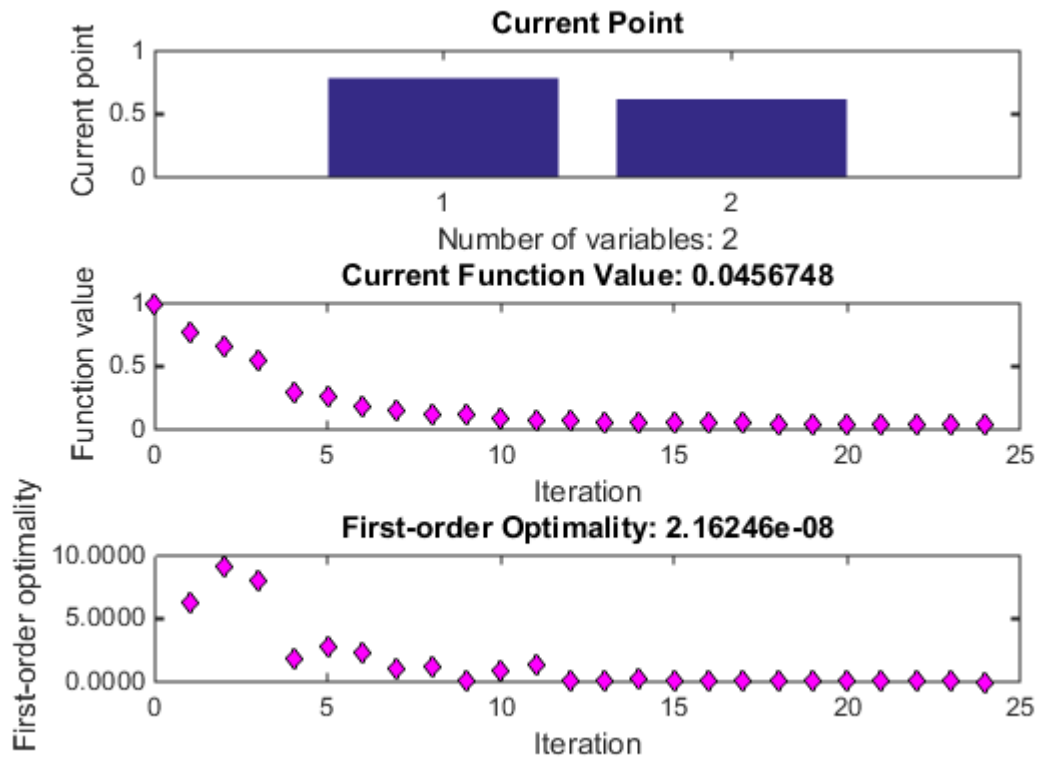
- The "Current Point" plot graphically shows the minimizer [0.786,0.618], which is reported as the **Final point** in the **Run solver and view results** pane. This plot updates at each iteration, showing the intermediate iterates.

- The "Current Function Value" plot shows the objective function value at all iterations. This graph is nearly monotone, showing fmincon reduces the objective function at almost every iteration.

- The "First-order Optimality" plot shows the first-order optimality measure at all iterations.

Running the Optimization from the Command Line

1 Write the nonlinear objective and constraint functions, including the derivatives, as shown in "Running the Optimization Using the Optimization App" on page 3-29.

2 Create an options structure that includes calling the three plot functions:

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
 'GradObj','on','GradConstr','on','PlotFcns',{@optimplotx,...
   @optimplotfval,@optimplotfirstorderopt});
```

3 Call `fmincon`:

```
x = fmincon(@rosenboth,[O O],[],[],[],[],[],[],...
   @unitdisk2,options)
```

4 `fmincon` gives the following output:

```
Local minimum found that satisfies the constraints.

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

x =
   0.7864    0.6177
```

`fmincon` also displays the three plot functions, shown at the end of "Running the Optimization Using the Optimization App" on page 3-29.

Output Functions

What Is an Output Function?

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 13-20 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 app for a solver.

Example: Using Output Functions

- "What the Example Contains" on page 3-35
- "Writing the Output Function" on page 3-36
- "Writing the Example Function File" on page 3-37
- "Running the Example" on page 3-38

What the Example Contains

The following example continues the one in "Nonlinear Inequality Constraints" on page 6-45, which calls the function `fmincon` at the command line to solve a nonlinear, constrained optimization problem. The example in this section uses a function file to call `fmincon`. The 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 file is here: "Writing the Example Function File" on page 3-37.

Writing the Output Function

You specify the output function in `options`, such as

```
options = optimoptions(@fmincon,'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 function of the function file described in "Writing the Example Function File" on page 3-37. 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.
            % Add .15 to x(1) to separate label from plotted 'o'
            text(x(1)+.15,x(2),num2str(optimValues.iteration));
        case 'done'
            hold off
```

```
        otherwise
    end
end
```

See " Using Handles to Store Function Parameters" 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 13-20.

Writing the Example Function File

To create the function file for this example:

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

```
function [history,searchdir] = runfmincon

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

% call optimization
x0 = [-1 1];
options = optimoptions(@fmincon,'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.
            % Add .15 to x(1) to separate label from plotted 'o'
                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 folder 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
Local minimum found that satisfies the constraints.

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

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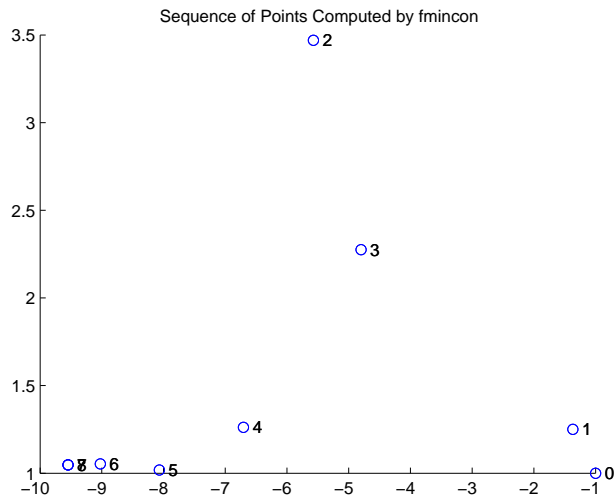
The optimal point occurs at the eighth iteration. Note that the last two points in the sequence are so close that they overlap.

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
```

4

Steps to Take After Running a Solver

Overview of Next Steps

This topic addresses questions you might have after running a solver. The questions include:

- Is the answer reliable?
- What can you do if the solver fails?
- Is the minimum smaller than all other minima, or only smaller than nearby minima? ("Local vs. Global Optima" on page 4-26)
- What can you do if the solver takes too long?

The list of questions is not exhaustive. It covers common or basic problems.

You can access relevant answers from many solvers' default exit message. The first line of the exit message contains a link to a brief description of the result. This description contains a link leading to documentation.

When the Solver Fails

In this section...

Too Many Iterations or Function Evaluations

The solver stopped because it reached a limit on the number of iterations or function evaluations before it minimized the objective to the requested tolerance. To proceed, try one or more of the following.

1. Enable Iterative Display

Set the `Display` option to `'iter'`. This setting shows the results of the solver iterations.

To enable iterative display:

- Using the Optimization app, choose **Level of display** to be `iterative` or `iterative with detailed message`.
- At the MATLAB command line, enter

  ```
  options = optimoptions('solvername','Display','iter');
  ```

  Call the solver using the `options` structure.

For an example of iterative display, see "Interpreting the Result" on page 1-10.

What to Look For in Iterative Display

- See if the objective function (`Fval` or `f(x)` or `Resnorm`) decreases. Decrease indicates progress.

- Examine constraint violation (`Max constraint`) to ensure that it decreases towards `0`. Decrease indicates progress.

- See if the first-order optimality decreases towards `0`. Decrease indicates progress.

- See if the `Trust-region radius` decreases to a small value. This decrease indicates that the objective might not be smooth.

What to Do

- If the solver seemed to progress:

    1 Set `MaxIter` and/or `MaxFunEvals` to values larger than the defaults. You can see the default values in the Optimization app, or in the Options table in the solver's function reference pages.

    2 Start the solver from its last calculated point.

- If the solver is not progressing, try the other listed suggestions.

2. Relax Tolerances

If `TolX` or `TolFun`, for example, are too small, the solver might not recognize when it has reached a minimum; it can make futile iterations indefinitely.

To change tolerances using the Optimization app, use the **Stopping criteria** list at the top of the **Options** pane.

To change tolerances at the command line, use `optimoptions` as described in "Set and Change Options" on page 2-56.

The `DiffMaxChange` and `DiffMinChange` options can affect a solver's progress. These options control the step size in finite differencing for derivative estimation.

3. Start the Solver From Different Points

See Change the Initial Point.

4. Check Objective and Constraint Function Definitions

For example, check that your objective and nonlinear constraint functions return the correct values at some points. See Check your Objective and Constraint Functions. Check that an infeasible point does not cause an error in your functions; see "Iterations Can Violate Constraints" on page 2-32.

5. Center and Scale Your Problem

Solvers run more reliably when each coordinate has about the same effect on the objective and constraint functions. Multiply your coordinate directions with appropriate scalars to equalize the effect of each coordinate. Add appropriate values to certain coordinates to equalize their size.

Example: Centering and Scaling

Consider minimizing `1e6*x(1)^2 + 1e-6*x(2)^2`:

```
f = @(x) 10^6*x(1)^2 + 10^-6*x(2)^2;
```

Minimize f using the `fminunc` `'quasi-newton'` algorithm:

```
opts = optimoptions('fminunc','Display','none','Algorithm','quasi-newton');
x = fminunc(f,[0.5;0.5],opts)

x =
          0
     0.5000
```

The result is incorrect; poor scaling interfered with obtaining a good solution.

Scale the problem. Set

```
D = diag([1e-3,1e3]);
fr = @(y) f(D*y);
y = fminunc(fr, [0.5;0.5], opts)

y =
      0
      0 % the correct answer
```

Similarly, poor centering can interfere with a solution.

```
fc = @(z)fr([z(1)-1e6;z(2)+1e6]); % poor centering
z = fminunc(fc,[.5 .5],opts)

z =
  1.0e+005 *
   10.0000  -10.0000 % looks good, but...

z - [1e6 -1e6] % checking how close z is to 1e6

ans =
```

```
    -0.0071    0.0078 % reveals a distance

fcc = @(w)fc([w(1)+1e6;w(2)-1e6]); % centered

w = fminunc(fcc,[.5 .5],opts)

w =
     0     0 % the correct answer
```

6. Provide Gradient or Jacobian

If you do not provide gradients or Jacobians, solvers estimate gradients and Jacobians by finite differences. Therefore, providing these derivatives can save computational time, and can lead to increased accuracy.

For constrained problems, providing a gradient has another advantage. A solver can reach a point x such that x is feasible, but finite differences around x always lead to an infeasible point. In this case, a solver can fail or halt prematurely. Providing a gradient allows a solver to proceed.

Provide gradients or Jacobians in the files for your objective function and nonlinear constraint functions. For details of the syntax, see "Writing Scalar Objective Functions" on page 2-18, "Writing Vector and Matrix Objective Functions" on page 2-25, and "Nonlinear Constraints" on page 2-35.

To check that your gradient or Jacobian function is correct, use the `DerivativeCheck` option, as described in "Checking Validity of Gradients or Jacobians" on page 2-65.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

For examples using gradients and Jacobians, see "Minimization with Gradient and Hessian" on page 6-15, "Nonlinear Constraints with Gradients" on page 6-47, "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80, "Nonlinear Equations with Analytic Jacobian" on page 11-9, and "Nonlinear Equations with Jacobian" on page 11-14.

7. Provide Hessian

Solvers often run more reliably and with fewer iterations when you supply a Hessian.

The following solvers and algorithms accept Hessians:

- `fmincon` `interior-point`. Write the Hessian as a separate function. For an example, see "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50.
- `fmincon` `trust-region-reflective`. Give the Hessian as the third output of the objective function. For an example, see "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75.
- `fminunc` `trust-region`. Give the Hessian as the third output of the objective function. For an example, see "Minimization with Gradient and Hessian" on page 6-15.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

Converged to an Infeasible Point

Usually, you get this result because the solver was unable to find a point satisfying all constraints to within the `TolCon` constraint tolerance. However, the solver might have located or started at a feasible point, and converged to an infeasible point. If the solver lost feasibility, see "Solver Lost Feasibility" on page 4-8.

To proceed when the solver found no feasible point, try one or more of the following.
"1. Check Linear Constraints" on page 4-7
"2. Check Nonlinear Constraints" on page 4-8

1. Check Linear Constraints

Try finding a point that satisfies the bounds and linear constraints by solving a linear programming problem.

1 Define a linear programming problem with an objective function that is always zero:

```
f = zeros(size(x0)); % assumes x0 is the initial point
```
2 Solve the linear programming problem to see if there is a feasible point:

```
xnew = linprog(f,A,b,Aeq,beq,lb,ub);
```
3 If there is a feasible point xnew, use xnew as the initial point and rerun your original problem.
4 If there is no feasible point, your problem is not well-formulated. Check the definitions of your bounds and linear constraints.

2. Check Nonlinear Constraints

After ensuring that your bounds and linear constraints are feasible (contain a point satisfying all constraints), check your nonlinear constraints.

*   Set your objective function to zero:

    ```
    @(x)0
    ```

    Run your optimization with all constraints and with the zero objective. If you find a feasible point xnew, set x0 = xnew and rerun your original problem.

*   If you do not find a feasible point using a zero objective function, use the zero objective function with several initial points.

    *   If you find a feasible point xnew, set x0 = xnew and rerun your original problem.
    *   If you do not find a feasible point, try relaxing the constraints, discussed next.

Try relaxing your nonlinear inequality constraints, then tightening them.

1   Change the nonlinear constraint function c to return c-Δ, where Δ is a positive number. This change makes your nonlinear constraints easier to satisfy.
2   Look for a feasible point for the new constraint function, using either your original objective function or the zero objective function.

    1   If you find a feasible point,

        a   Reduce Δ
        b   Look for a feasible point for the new constraint function, starting at the previously found point.

    2   If you do not find a feasible point, try increasing Δ and looking again.

If you find no feasible point, your problem might be truly infeasible, meaning that no solution exists. Check all your constraint definitions again.

Solver Lost Feasibility

If the solver started at a feasible point, but converged to an infeasible point, try the following techniques.

*   Try a different algorithm. The fmincon 'sqp' and 'interior-point' algorithms are usually the most robust, so try one or both of them first.

- Tighten the bounds. Give the highest `lb` and lowest `ub` vectors that you can. This can help the solver to maintain feasibility. The fmincon `'sqp'` and `'interior-point'` algorithms obey bounds at every iteration, so tight bounds help throughout the optimization.

Problem Unbounded

The solver reached a point whose objective function was less than the objective limit tolerance.

- Your problem might be truly unbounded. In other words, there is a sequence of points x_i with
$\lim f(x_i) = -\infty$.

  and such that all the x_i satisfy the problem constraints.
- Check that your problem is formulated correctly. Solvers try to minimize objective functions; if you want a maximum, change your objective function to its negative. For an example, see "Maximizing an Objective" on page 2-28.
- Try scaling or centering your problem. See Center and Scale Your Problem.
- Relax the objective limit tolerance by using `optimoptions` to reduce the value of the `ObjectiveLimit` tolerance.

fsolve Could Not Solve Equation

`fsolve` can fail to solve an equation for various reasons. Here are some suggestions for how to proceed:

1 Try Changing the Initial Point. `fsolve` relies on an initial point. By giving it different initial points, you increase the chances of success.

2 Check the definition of the equation to make sure that it is smooth. `fsolve` might fail to converge for equations with discontinuous gradients, such as absolute value. `fsolve` can fail to converge for functions with discontinuities.

3 Check that the equation is "square," meaning equal dimensions for input and output (has the same number of unknowns as values of the equation).

4 Change tolerances, especially `TolFun` and `TolX`. If you attempt to get high accuracy by setting tolerances to very small values, `fsolve` can fail to converge. If you set tolerances that are too high, `fsolve` can fail to solve an equation accurately.

5 Check the problem definition. Some problems have no real solution, such as
`x^2 + 1 = 0`.

Solver Takes Too Long

Solvers can take excessive time for various reasons. To diagnose the reason, use one or more of the following techniques.

1. "Enable Iterative Display" on page 4-11
2. "Enable FunValCheck" on page 4-11
3. "Use Appropriate Tolerances" on page 4-12
4. "Use a Plot Function" on page 4-12
5. "Enable DerivativeCheck" on page 4-12
6. "Use Inf Instead of a Large, Arbitrary Bound" on page 4-13
7. "Use an Output Function" on page 4-13
8. "Use a Sparse Solver or a Multiply Function" on page 4-13
9. "Use Parallel Computing" on page 4-14

Enable Iterative Display

Set the `Display` option to `'iter'`. This setting shows the results of the solver iterations.

To enable iterative display:

- Using the Optimization app, choose **Level of display** to be `iterative` or `iterative with detailed message`.
- At the MATLAB command line, enter

  ```
  options = optimoptions('solvername','Display','iter');
  ```

  Call the solver using the `options` structure.

For an example of iterative display, see "Interpreting the Result" on page 1-10. For more information, see "What to Look For in Iterative Display" on page 4-3.

Enable FunValCheck

Sometimes a solver fails because an objective function or nonlinear constraint function returns a complex value, infinity, or NaN. To halt solver iterations in these cases, enable the `FunValCheck` option.

- Using the Optimization app, check the box labeled **Error if user-supplied function returns Inf, NaN, or complex** in the **Function value check** pane.
- At the MATLAB command line, enter

  ```
  options = optimoptions('solvername','FunValCheck','on');
  ```

  Call the solver using the `options` structure.

Use Appropriate Tolerances

Solvers can fail to converge if tolerances are too small, especially `TolFun` and `TolX`.

To change tolerances using the Optimization app, use the **Stopping criteria** list at the top of the **Options** pane.

To change tolerances at the command line, use `optimoptions` as described in "Set and Change Options" on page 2-56.

Use a Plot Function

You can obtain more visual or detailed information about solver iterations using a plot function. For a list of the predefined plot functions, see **Options > Plot functions** in the Optimization app. The Options section of your solver's function reference pages also lists the plot functions.

To use a plot function:

- Using the Optimization app, check the boxes next to each plot function you wish to use.
- At the MATLAB command line, enter

  ```
  options = optimoptions('solvername','PlotFcns',{@plotfcn1,@plotfcn2,...});
  ```

  Call the solver using the `options` structure.

For an example of using a plot function, see "Using a Plot Function" on page 3-29.

Enable DerivativeCheck

If you have supplied derivatives (gradients or Jacobians) to your solver, the solver can fail to converge if the derivatives are inaccurate. For more information about using the

`DerivativeCheck` option, see "Checking Validity of Gradients or Jacobians" on page 2-65.

Use Inf Instead of a Large, Arbitrary Bound

If you use a large, arbitrary bound (upper or lower), a solver can take excessive time, or even fail to converge. However, if you set `Inf` or `-Inf` as the bound, the solver can take less time, and might converge better.

Why? An interior-point algorithm can set an initial point to the midpoint of finite bounds. Or an interior-point algorithm can try to find a "central path" midway between finite bounds. Therefore, a large, arbitrary bound can resize those components inappropriately. In contrast, infinite bounds are ignored for these purposes.

Minor point: Some solvers use memory for each constraint, primarily via a constraint Hessian. Setting a bound to `Inf` or `-Inf` means there is no constraint, so there is less memory in use, because a constraint Hessian has lower dimension.

Use an Output Function

You can obtain detailed information about solver iterations using an output function. Solvers call output functions at each iteration. You write output functions using the syntax described in "Output Function" on page 13-20.

For an example of using an output function, see "Example: Using Output Functions" on page 3-35.

Use a Sparse Solver or a Multiply Function

Large problems can cause MATLAB to run out of memory or time. Here are some suggestions for using less memory:

- Use a large-scale algorithm if possible (see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12). These algorithms include `trust-region-reflective`, `interior-point`, the `fminunc trust-region` algorithm, the `fsolve trust-region-dogleg` algorithm, and the `Levenberg-Marquardt` algorithm. In contrast, the `active-set`, `quasi-newton`, and `sqp` algorithms are not large-scale.

  Use sparse matrices for your linear constraints.

- Use a Jacobian multiply function or Hessian multiply function. For examples, see "Jacobian Multiply Function with Linear Least Squares" on page 10-26, "Quadratic Minimization with Dense, Structured Hessian" on page 9-19, and "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75.

Use Parallel Computing

If you have a Parallel Computing Toolbox license, your solver might run faster using parallel computing. For more information, see "Parallel Computing".

When the Solver Might Have Succeeded

In this section...
"Final Point Equals Initial Point" on page 4-15
"Local Minimum Possible" on page 4-15

Final Point Equals Initial Point

The initial point seems to be a local minimum or solution because the first-order optimality measure is close to 0. You might be unhappy with this result, since the solver did not improve your initial point.

If you are unsure that the initial point is truly a local minimum, try:

1   Starting from different points — see Change the Initial Point.
2   Checking that your objective and constraints are defined correctly (for example, do they return the correct values at some points?) — see Check your Objective and Constraint Functions. Check that an infeasible point does not cause an error in your functions; see "Iterations Can Violate Constraints" on page 2-32.
3   Changing tolerances, such as `TolFun`, `TolCon`, and `TolX` — see Use Appropriate Tolerances.
4   Scaling your problem so each coordinate has about the same effect — see Rescale the Problem.
5   Providing gradient and Hessian information — see Provide Analytic Gradients or Jacobian and Provide a Hessian.

Local Minimum Possible

The solver might have reached a local minimum, but cannot be certain because the first-order optimality measure is not less than the `TolFun` tolerance. (To learn more about first-order optimality measure, see "First-Order Optimality Measure" on page 3-11.) To see if the reported solution is reliable, consider the following suggestions.
"1. Nonsmooth Functions" on page 4-16
"2. Rerun Starting At Final Point" on page 4-16
"3. Try a Different Algorithm" on page 4-17
"4. Change Tolerances" on page 4-19
"5. Rescale the Problem" on page 4-20
"6. Check Nearby Points" on page 4-20

1. Nonsmooth Functions

If you try to minimize a nonsmooth function, or have nonsmooth constraints, "Local Minimum Possible" can be the best exit message. This is because the first-order optimality conditions do not apply at a nonsmooth point.

To satisfy yourself that the solution is adequate, try to Check Nearby Points.

2. Rerun Starting At Final Point

Restarting an optimization at the final point can lead to a solution with a better first-order optimality measure. A better (lower) first-order optimality measure gives you more reason to believe that the answer is reliable.

For example, consider the following minimization problem, taken from an example (echodemo symbolic_optim_demo):

```
options = optimoptions('fminunc','Algorithm','quasi-newton');
funh = @(x)log(1 + (x(1) - 4/3)^2 + 3*(x(2) - (x(1)^3 - x(1)))^2);
[xfinal fval exitflag] = fminunc(funh,[-1;2],options)

Local minimum possible.

fminunc stopped because it cannot decrease the
objective function along the current search direction.

xfinal =
    1.3333
    1.0370

fval =
  8.5265e-014

exitflag =
     5
```

The exit flag value of **5** indicates that the first-order optimality measure was above the function tolerance. Run the minimization again starting from xfinal:

```
[xfinal2 fval2 exitflag2] = fminunc(funh,xfinal,options)
```

```
Local minimum found.

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

xfinal2 =
    1.3333
    1.0370

fval2 =
  6.5281e-014

exitflag2 =
     1
```

The local minimum is "found," not "possible," and the exitflag is 1, not 5. The two solutions are virtually identical. Yet the second run has a more satisfactory exit message, since the first-order optimality measure was low enough: 7.5996e-007, instead of 3.9674e-006.

3. Try a Different Algorithm

Many solvers give you a choice of algorithm. Different algorithms can lead to the use of different stopping criteria.

For example, Rerun Starting At Final Point returns exitflag 5 from the first run. This run uses the quasi-newton algorithm.

The trust-region algorithm requires a user-supplied gradient. betopt.m contains a calculation of the objective function and gradient:

```
function [f gradf] = betopt(x)

g = 1 + (x(1)-4/3)^2 + 3*(x(2) - (x(1)^3-x(1)))^2;
f = log(g);
gradf(1) = 2*(x(1)-4/3) + 6*(x(2) - (x(1)^3-x(1)))*(1-3*x(1)^2);
gradf(1) = gradf(1)/g;
gradf(2) = 6*(x(2) - (x(1)^3 -x(1)))/g;
```

Running the optimization using the trust-region algorithm results in a different exitflag:

```
options = optimoptions('fminunc','Algorithm','trust-region','GradObj','on');
```

```
[xfinal3 fval3 exitflag3] = fminunc(@betopt,[-1;2],options)

Local minimum possible.

fminunc stopped because the final change in function value
relative to its initial value is less than the default value
of the function tolerance.

xfinal3 =
    1.3333
    1.0370

fval3 =
  7.6659e-012

exitflag3 =
     3
```

The exit condition is better than the `quasi-newton` condition, though it is still not the best. Rerunning the algorithm from the final point produces a better point, with extremely small first-order optimality measure:

```
[xfinal4 fval4 eflag4 output4] = fminunc(@betopt,xfinal3,options)

Local minimum found.

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

xfinal4 =

    1.3333
    1.0370

fval4 =
     0

eflag4 =
     1

output4 =
         iterations: 1
          funcCount: 2
        cgiterations: 1
```

```
       firstorderopt: 7.5497e-11
            algorithm: 'trust-region'
              message: 'Local minimum found.

Optimization completed because the size o...'
        constrviolation: []
```

4. Change Tolerances

Sometimes tightening or loosening tolerances leads to a more satisfactory result. For example, choose a smaller value of TolFun in the Try a Different Algorithm section:

```
options = optimoptions('fminunc','Algorithm','trust-region',...
    'TolFun',1e-8,'GradObj','on'); % default=1e-6
[xfinal3 fval3 eflag3 output3] = fminunc(@betopt,[-1;2],options)

Local minimum found.

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

xfinal3 =
    1.3333
    1.0370

fval3 =
     0

eflag3 =
     1

output3 =
          iterations: 15
           funcCount: 16
         cgiterations: 12
       firstorderopt: 7.5497e-11
            algorithm: 'trust-region'
              message: 'Local minimum found.

Optimization completed because the size...'
        constrviolation: []
```

fminunc took one more iteration than before, arriving at a better solution.

5. Rescale the Problem

Try to have each coordinate give about the same effect on the objective and constraint functions by scaling and centering. For examples, see Center and Scale Your Problem.

6. Check Nearby Points

Evaluate your objective function and constraints, if they exist, at points near the final point. If the final point is a local minimum, nearby feasible points have larger objective function values. See Check Nearby Points for an example.

If you have a Global Optimization Toolbox license, try running the `patternsearch` solver from the final point. `patternsearch` examines nearby points, and accepts all types of constraints.

7. Change Finite Differencing Options

Central finite differences are more time-consuming to evaluate, but are much more accurate. Use central differences when your problem can have high curvature.

To choose central differences at the command line, use `optimoptions` to set `'FinDiffType'` to `'central'`, instead of the default `'forward'`.

To choose central differences in the Optimization app, set **Options > Approximated derivatives > Type** to be `central differences`.

8. Provide Analytic Gradients or Jacobian

If you do not provide gradients or Jacobians, solvers estimate gradients and Jacobians by finite differences. Therefore, providing these derivatives can save computational time, and can lead to increased accuracy.

For constrained problems, providing a gradient has another advantage. A solver can reach a point `x` such that `x` is feasible, but finite differences around `x` always lead to an infeasible point. In this case, a solver can fail or halt prematurely. Providing a gradient allows a solver to proceed.

Provide gradients or Jacobians in the files for your objective function and nonlinear constraint functions. For details of the syntax, see "Writing Scalar Objective Functions" on page 2-18, "Writing Vector and Matrix Objective Functions" on page 2-25, and "Nonlinear Constraints" on page 2-35.

To check that your gradient or Jacobian function is correct, use the `DerivativeCheck` option, as described in "Checking Validity of Gradients or Jacobians" on page 2-65.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

For examples using gradients and Jacobians, see "Minimization with Gradient and Hessian" on page 6-15, "Nonlinear Constraints with Gradients" on page 6-47, "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80, "Nonlinear Equations with Analytic Jacobian" on page 11-9, and "Nonlinear Equations with Jacobian" on page 11-14.

9. Provide a Hessian

Solvers often run more reliably and with fewer iterations when you supply a Hessian.

The following solvers and algorithms accept Hessians:

- `fmincon` `interior-point`. Write the Hessian as a separate function. For an example, see "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50.
- `fmincon` `trust-region-reflective`. Give the Hessian as the third output of the objective function. For an example, see "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75.
- `fminunc` `trust-region`. Give the Hessian as the third output of the objective function. For an example, see "Minimization with Gradient and Hessian" on page 6-15.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

The example in "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80 shows `fmincon` taking 77 iterations without a Hessian, but only 19 iterations with a Hessian.

When the Solver Succeeds

What Can Be Wrong If The Solver Succeeds?

A solver can report that a minimization succeeded, and yet the reported solution can be incorrect. For a rather trivial example, consider minimizing the function $f(x) = x^3$ for x between –2 and 2, starting from the point `1/3`:

```
options = optimoptions('fmincon','Algorithm','active-set');
ffun = @(x)x^3;
xfinal = fmincon(ffun,1/3,[],[],[],[],-2,2,[],options)

Local minimum found that satisfies the constraints.

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

No active inequalities.

xfinal =
 -1.5056e-008
```

The true minimum occurs at `x = -2`. `fmincon` gives this report because the function $f(x)$ is so flat near $x = 0$.

Another common problem is that a solver finds a local minimum, but you might want a global minimum. For more information, see "Local vs. Global Optima" on page 4-26.

Lesson: check your results, even if the solver reports that it "found" a local minimum, or "solved" an equation.

This section gives techniques for verifying results.

1. Change the Initial Point

The initial point can have a large effect on the solution. If you obtain the same or worse solutions from various initial points, you become more confident in your solution.

For example, minimize $f(x) = x^3 + x^4$ starting from the point 1/4:

```
ffun = @(x)x^3 + x^4;
options = optimoptions('fminunc','Algorithm','quasi-newton');
[xfinal fval] = fminunc(ffun,1/4,options)

Local minimum found.

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

x =
 -1.6764e-008

fval =
 -4.7111e-024
```

Change the initial point by a small amount, and the solver finds a better solution:

```
[xfinal fval] = fminunc(ffun,1/4+.001,options)

Local minimum found.

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

xfinal =
   -0.7500

fval =
   -0.1055
```

`x = -0.75` is the global solution; starting from other points cannot improve the solution.

For more information, see "Local vs. Global Optima" on page 4-26.

2. Check Nearby Points

To see if there are better values than a reported solution, evaluate your objective function and constraints at various nearby points.

For example, with the objective function `ffun` from "What Can Be Wrong If The Solver Succeeds?" on page 4-22, and the final point `xfinal = -1.5056e-008`, calculate `ffun(xfinal±Δ)` for some Δ:

```
delta = .1;
[ffun(xfinal),ffun(xfinal+delta),ffun(xfinal-delta)]

ans =
    -0.0000    0.0011    -0.0009
```

The objective function is lower at `ffun(xfinal-Δ)`, so the solver reported an incorrect solution.

A less trivial example:

```
options = optimoptions(@fmincon,'Algorithm','active-set');
lb = [0,-1]; ub = [1,1];
ffun = @(x)(x(1)-(x(1)-x(2))^2);
[x fval exitflag] = fmincon(ffun,[1/2 1/3],[],[],[],[],...
                            lb,ub,[],options)

Local minimum found that satisfies the constraints.

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

Active inequalities (to within options.TolCon = 1e-006):
  lower      upper      ineqlin    ineqnonlin
    1

x =
  1.0e-007 *
        0    0.1614

fval =
 -2.6059e-016
```

```
exitflag =
     1
```

Evaluating `ffun` at nearby feasible points shows that the solution `x` is not a true minimum:

```
[ffun([0,.001]),ffun([0,-.001]),...
    ffun([.001,-.001]),ffun([.001,.001])]

ans =
  1.0e-003 *
   -0.0010   -0.0010    0.9960    1.0000
```

The first two listed values are smaller than the computed minimum `fval`.

If you have a Global Optimization Toolbox license, you can use the `patternsearch` function to check nearby points.

3. Check your Objective and Constraint Functions

Double-check your objective function and constraint functions to ensure that they correspond to the problem you intend to solve. Suggestions:

- Check the evaluation of your objective function at a few points.
- Check that each inequality constraint has the correct sign.
- If you performed a maximization, remember to take the negative of the reported solution. (This advice assumes that you maximized a function by minimizing the negative of the objective.) For example, to maximize $f(x) = x - x^2$, minimize $g(x) = -x + x^2$:

  ```
  options = optimoptions('fminunc','Algorithm','quasi-newton');
  [x fval] = fminunc(@(x)-x+x^2,0,options)

  Local minimum found.

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

  x =
      0.5000

  fval =
     -0.2500
  ```

The maximum of *f* is 0.25, the negative of `fval`.

- Check that an infeasible point does not cause an error in your functions; see "Iterations Can Violate Constraints" on page 2-32.

Local vs. Global Optima

- "Why Didn't the Solver Find the Smallest Minimum?" on page 4-26
- "Searching for a Smaller Minimum" on page 4-27
- "Basins of Attraction" on page 4-27

Why Didn't the Solver Find the Smallest Minimum?

In general, solvers return a local minimum. The result might be a global minimum, but there is no guarantee that it is. This section describes why solvers behave this way, and gives suggestions for ways to search for a global minimum, if needed.

- A *local* minimum of a function is a point where the function value is smaller than at nearby points, but possibly greater than at a distant point.
- A *global* minimum is a point where the function value is smaller than at all other feasible points.
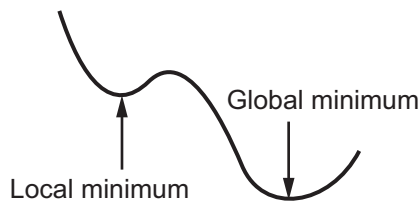
Generally, Optimization Toolbox solvers find a local optimum. (This local optimum can be a global optimum.) They find the optimum in the *basin of attraction* of the starting point. For more information about basins of attraction, see "Basins of Attraction" on page 4-27.

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.

- Global Optimization Toolbox functions, such as `simulannealbnd`, attempt to search more than one basin of attraction.

Searching for a Smaller Minimum

If you need a global optimum, you must find an initial value for your solver in the basin of attraction of a global optimum.

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. Use points drawn 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.
- If you have a Global Optimization Toolbox license, use the GlobalSearch or MultiStart solvers. These solvers automatically generate random start points within bounds.

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

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. The *basin of attraction* for steepest descent is the set of initial values that lead to the same local minimum.

The following figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and shows directions of steepest descent

with arrows. For this and subsequent figures, black dots represent local minima. Every steepest descent path, starting at a point $x(0)$, goes to the black dot in the basin containing $x(0)$.
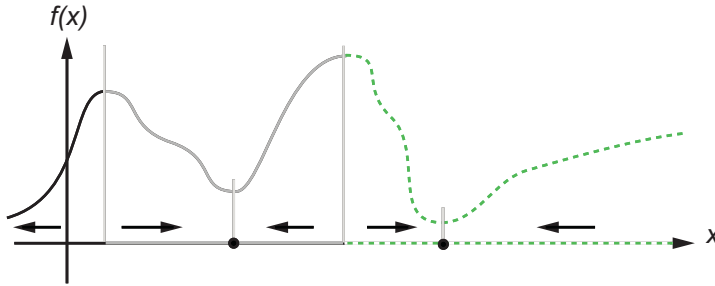
One-dimensional basins

The following figure shows how steepest descent paths can be more complicated in more dimensions.
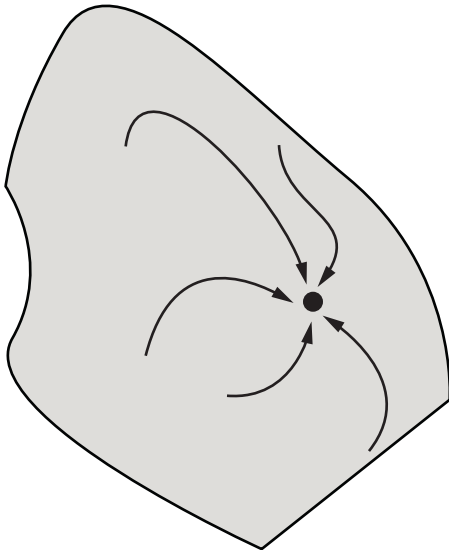
One basin of attraction, showing steepest descent paths from various starting points

The following figure shows even more complicated paths and basins of attraction.
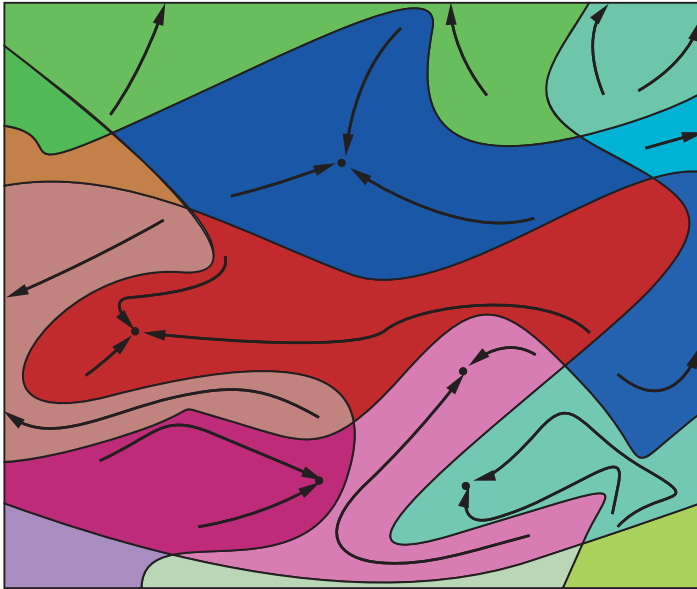
Several basins of attraction

Constraints can break up one basin of attraction into several pieces. For example, consider minimizing y subject to:

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

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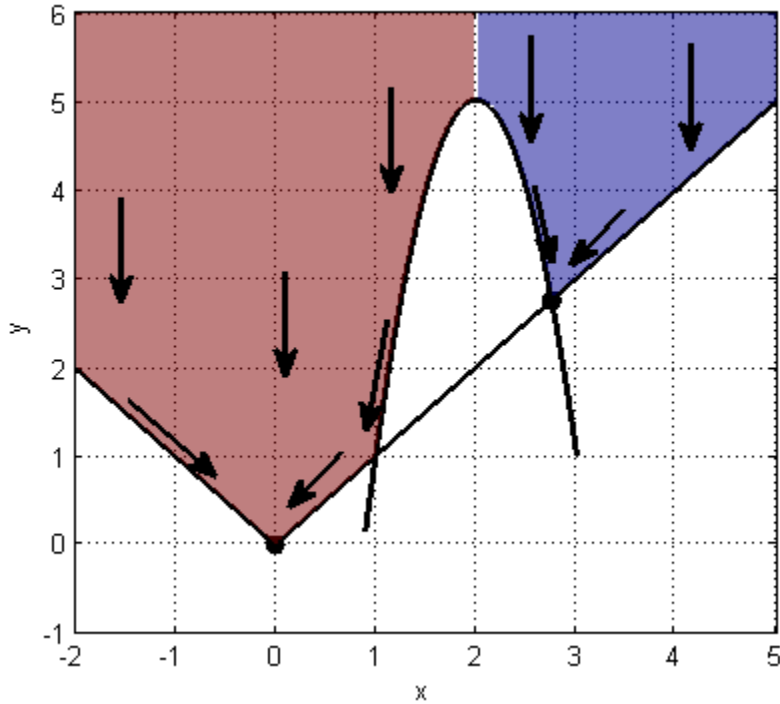
The steepest descent paths are straight lines down to the constraint boundaries. From the constraint boundaries, the steepest descent paths travel down along the boundaries. The final point is either (0,0) or (11/4,11/4), depending on whether the initial *x*-value is above or below 2.

Optimizing a Simulation or Ordinary Differential Equation

What Is Optimizing a Simulation or ODE?

Sometimes your objective function or nonlinear constraint function values are available only by simulation or by numerical solution of an ordinary differential equation (ODE). Such optimization problems have several common characteristics and challenges, discussed in "Potential Problems and Solutions" on page 4-31.

To optimize a Simulink® model easily, try using Simulink Design Optimization™.

Potential Problems and Solutions

- "Problems in Finite Differences" on page 4-31
- "Suggestions for Finite Differences" on page 4-32
- "Problems in Stochastic Functions" on page 4-35
- "Suggestions for Stochastic Functions" on page 4-35
- "Common Calculation of Objective and Constraints" on page 4-35
- "Failure in Objective or Constraint Function Evaluation" on page 4-35
- "Suggestions for Evaluation Failures" on page 4-35

Problems in Finite Differences

Optimization Toolbox solvers use derivatives of objective and constraint functions internally. By default, they estimate these derivatives using finite difference approximations of the form

$$\frac{F(x + \delta) - F(x)}{\delta}$$

or

$$\frac{F(x+\delta) - F(x-\delta)}{2\delta}.$$

These finite difference approximations can be inaccurate because:

- A large value of δ allows more nonlinearity to affect the finite difference.
- A small value of δ leads to inaccuracy due to limited precision in numerics.

Specifically, for simulations and numerical solutions of ODEs:

- Simulations are often insensitive to small changes in parameters. This means that if you use too small a perturbation δ, the simulation can return a spurious estimated derivative of 0.
- Both simulations and numerical solutions of ODEs can have inaccuracies in their function evaluations. These inaccuracies can be amplified in finite difference approximations.
- Numerical solution of ODEs introduces noise at values much larger than machine precision. This noise can be amplified in finite difference approximations.
- If an ODE solver uses variable step sizes, then sometimes the number of ODE steps in the evaluation of $F(x + \delta)$ can differ from the number of steps in the evaluation of $F(x)$. This difference can lead to a spurious difference in the returned values, giving a misleading estimate of the derivative.

Suggestions for Finite Differences

- "Avoid Finite Differences by Using Direct Search" on page 4-32
- "Set Larger Finite Differences" on page 4-33
- "Use a Gradient Evaluation Function" on page 4-33
- "Use Tighter ODE Tolerances" on page 4-34

Avoid Finite Differences by Using Direct Search

If you have a Global Optimization Toolbox license, you can try using `patternsearch` as your solver. `patternsearch` does not attempt to estimate gradients, so does not suffer from the limitations in "Problems in Finite Differences" on page 4-31.

If you use `patternsearch` for expensive (time-consuming) function evaluations, use the `Cache` option:

```
options = psoptimset('Cache','on');
```

If you cannot use `patternsearch`, and have a relatively low-dimensional unconstrained minimization problem, try `fminsearch` instead. `fminsearch` does not use finite differences. However, `fminsearch` is not a fast or tunable solver.

Set Larger Finite Differences

You can sometimes avoid the problems in "Problems in Finite Differences" on page 4-31 by taking larger finite difference steps than the default.

*   If you have MATLAB R2011b or later, set the `FinDiffRelStep` option to a value larger than the default `sqrt(eps)` or `eps^(1/3)`, such as:

    *   For R2011b–R2012b:

        ```
        options = optimset('FinDiffRelStep',1e-3);
        ```
    *   For R2013a onwards:

        ```
        options = optimoptions('solvername','FinDiffRelStep',1e-3);
        ```

    If you have different scales in different components, set `FinDiffRelStep` to a vector proportional to the component scales.
*   If you have MATLAB R2011a or earlier, set the `DiffMinChange` option to a larger value than the default `1e-8`, and possibly set the `DiffMaxChange` option also, such as:

    ```
    options = optimset('DiffMinChange',1e-3,'DiffMaxChange',1);
    ```

Note: It is difficult to know how to set these finite difference sizes.

You can also try setting central finite differences:

```
options = optimoptions('solvername','FinDiffType','central');
```

Use a Gradient Evaluation Function

To avoid the problems of finite difference estimation, you can give an approximate gradient function in your objective and nonlinear constraints. Remember to set the

GradObj option to 'on' using optimoptions, and, if relevant, also set the GradConstr option to 'on'.

- For some ODEs, you can evaluate the gradient numerically at the same time as you solve the ODE. For example, suppose the differential equation for your objective function $z(t,x)$ is

$$\frac{d}{dt}z(t,x) = G(z,t,x),$$

where x is the vector of parameters over which you minimize. Suppose x is a scalar. Then the differential equation for its derivative y,

$$y(t,x) = \frac{d}{dx}z(t,x)$$

is

$$\frac{d}{dt}y(t,x) = \frac{\partial G(z,t,x)}{\partial z}y(t,x) + \frac{\partial G(z,t,x)}{\partial x},$$

where $z(t,x)$ is the solution of the objective function ODE. You can solve for $y(t,x)$ in the same system of differential equations as $z(t,x)$. This solution gives you an approximated derivative without ever taking finite differences. For nonscalar x, solve one ODE per component.

For theoretical and computational aspects of this method, see Leis and Kramer [2]. For computational experience with this and finite-difference methods, see Figure 7 of Raue et al. [3].

- For some simulations, you can estimate a derivative within the simulation. For example, the likelihood ratio technique described in Reiman and Weiss [4] or the infinitesimal perturbation analysis technique analyzed in Heidelberger, Cao, Zazanis, and Suri [1] estimate derivatives in the same simulation that estimates the objective or constraint functions.

Use Tighter ODE Tolerances

You can use odeset to set the AbsTol or RelTol ODE solver tolerances to values below their defaults. However, choosing too small a tolerance can lead to slow solutions, convergence failure, or other problems. Never choose a tolerance less than 1e-9 for

RelTol. The lower limit on each component of AbsTol depends on the scale of your problem, so there is no advice.

Problems in Stochastic Functions

If a simulation uses random numbers, then evaluating an objective or constraint function twice can return different results. This affects both function estimation and finite difference estimation. The value of a finite difference might be dominated by the variation due to randomness, rather than the variation due to different evaluation points x and $x + \delta$.

Suggestions for Stochastic Functions

If your simulation uses random numbers from a stream you control, reset the random stream before each evaluation of your objective or constraint functions. This practice can reduce the variability in results. For example, in an objective function:

```
function f = mysimulation(x)
rng default % or any other resetting method
...
end
```

For details, see "Generate Random Numbers That Are Repeatable".

Common Calculation of Objective and Constraints

Frequently, a simulation evaluates both the objective function and constraints during the same simulation run. Or, both objective and nonlinear constraint functions use the same expensive computation. Solvers such as fmincon separately evaluate the objective function and nonlinear constraint functions. This can lead to a great loss of efficiency, because the solver calls the expensive computation twice. To circumvent this problem, use the technique in "Objective and Nonlinear Constraints in the Same Function" on page 2-45.

Failure in Objective or Constraint Function Evaluation

Your simulation or ODE can fail for some parameter values.

Suggestions for Evaluation Failures

Set Appropriate Bounds

While you might not know all limitations on the parameter space, try to set appropriate bounds on all parameters, both upper and lower. This can speed up your optimization, and can help the solver avoid problematic parameter values.

Use a Solver That Respects Bounds

As described in "Iterations Can Violate Constraints" on page 2-32, some algorithms can violate bound constraints at intermediate iterations. For optimizing simulations and ODEs, use algorithms that always obey bound constraints. See "Algorithms That Satisfy Bound Constraints" on page 2-32.

Return NaN

If your simulation or ODE solver does not successfully evaluate an objective or nonlinear constraint function at a point *x*, have your function return `NaN`. Most Optimization Toolbox and Global Optimization Toolbox solvers have the robustness to attempt a different iterative step if they encounter a `NaN` value. These robust solvers include:

- `fmincon` `interior-point`, `sqp`, and `trust-region-reflective` algorithms
- `fminunc`
- `lsqcurvefit`
- `lsqnonlin`
- `patternsearch`

Some people are tempted to return an arbitrary large objective function value at an unsuccessful, infeasible, or other poor point. However, this practice can confuse a solver, because the solver does not realize that the returned value is arbitrary. When you return `NaN`, the solver can attempt to evaluate at a different point.

Bibliography

[1] Heidelberger, P., X.-R. Cao, M. A. Zazanis, and R. Suri. *Convergence properties of infinitesimal perturbation analysis estimates*. Management Science 34, No. 11, pp. 1281–1302, 1988.

[2] Leis, J. R., and Kramer, M.A. *The Simultaneous Solution and Sensitivity Analysis of Systems Described by Ordinary Differential Equations*. ACM Trans. Mathematical Software, Vol. 14, No. 1, pp. 45–60, 1988.

[3] Raue, A., et al. *Lessons Learned from Quantitative Dynamical Modeling in Systems Biology*. Available at http://www.plosone.org/article/info:doi/10.1371/journal.pone.0074335, 2013.

[4] Reiman, M. I., and A. Weiss. *Sensitivity analysis via likelihood ratios*. Proc. 18th Winter Simulation Conference, ACM, New York, pp. 285–289, 1986.

Optimization App

Optimization App

Note: The Optimization app warns that it will be removed in a future release.

In this section...
"Optimization App Basics" on page 5-2
"Specifying Certain Options" on page 5-8
"Importing and Exporting Your Work" on page 5-11

Optimization App Basics

- "How to Open the Optimization App" on page 5-2
- "Examples that Use the Optimization App" on page 5-4
- "Steps for Using the Optimization App" on page 5-4
- "Pausing and Stopping" on page 5-5
- "Viewing Results" on page 5-6
- "Final Point" on page 5-7
- "Starting a New Problem" on page 5-7

How to Open the Optimization App

To open the Optimization app, type

```
optimtool
```

in the Command Window. This opens the Optimization app, as shown in the following figure.
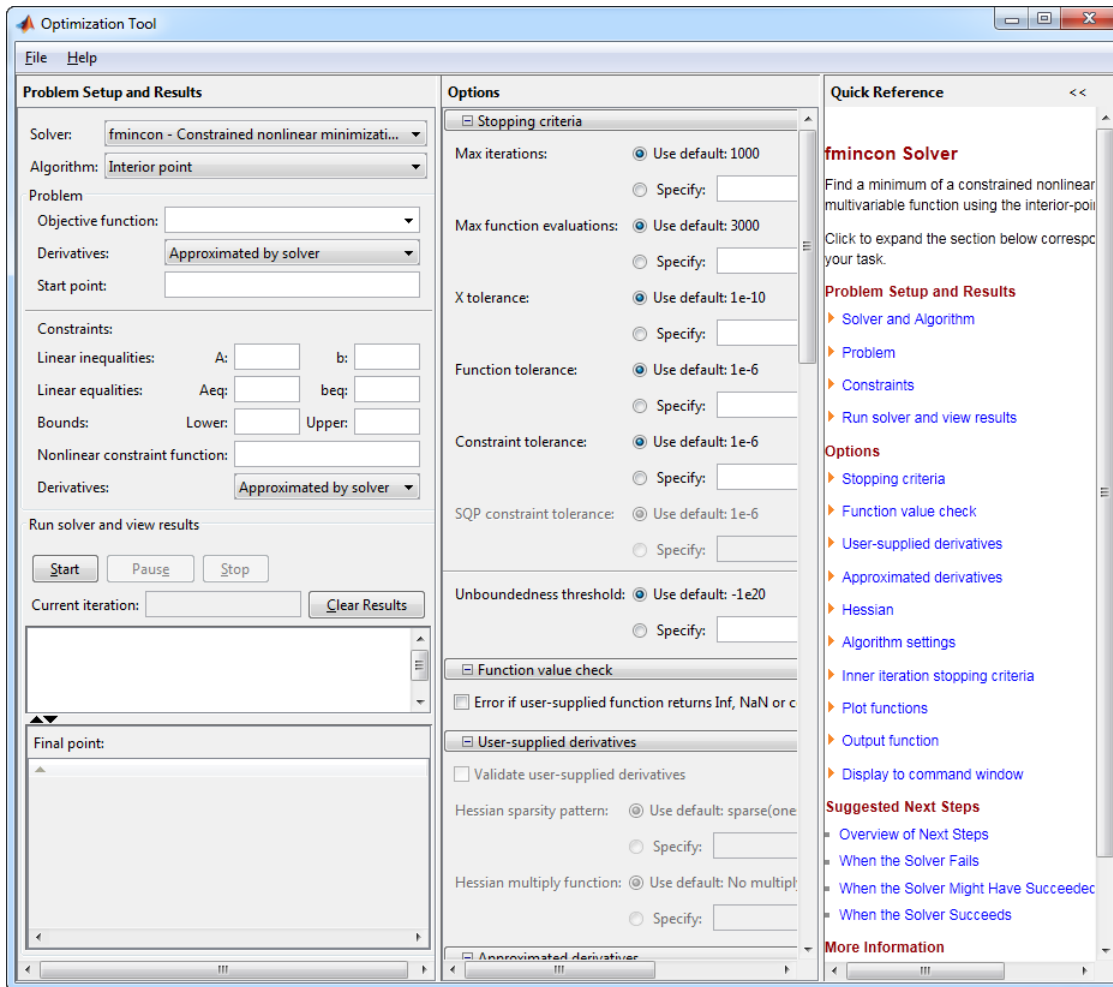
You can also start the Optimization app from the MATLAB **Apps** tab.
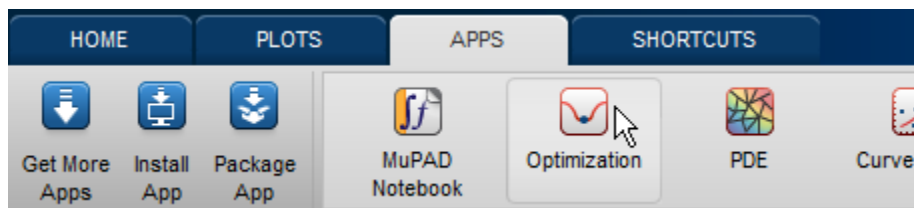
The reference page for the Optimization app provides variations for starting the `optimtool` function.

Examples that Use the Optimization App

The following documentation examples use the optimization app:

- "Solve a Constrained Nonlinear Problem" on page 1-3
- "Optimization App with the fmincon Solver" on page 6-63
- "Optimization App with the lsqlin Solver" on page 10-23
- "Plot Functions" on page 3-29
- "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50

Steps for Using the Optimization App

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

7. Import and export problem, options, and results

4. Specify options

1. Select solver and algorithm

2. Specify function to minimize

3. Set problem parameters for selected solver

5. Run solver

6. View solver status and results

Pausing and Stopping

While a solver 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 Your Work" on page 5-11.
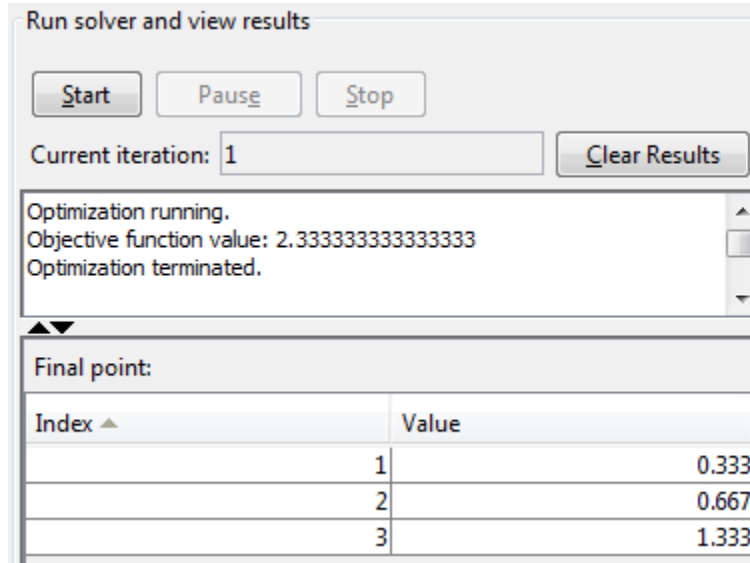
Viewing Results

When a solver 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**.

Sorting the Displayed Results

Depending on the solver and problem, results can be in the form of a table. If the table has multiple rows, sort the table by clicking a column heading. Click the heading again to sort the results in reverse.

For example, suppose you use the Optimization app to solve the `lsqlin` problem described in "Optimization App with the lsqlin Solver" on page 10-23. The result appears as follows.

To sort the results by value, from lowest to highest, click Value. The results were already in that order, so don't change.

To sort the results in reverse order, highest to lowest, click Value again.

Final point:	
Index	Value ▼
3	1.333
2	0.667
1	0.333

To return to the original order, click Index.

For an example of sorting a table returned by the Global Optimization Toolbox `gamultiobj` function, see "Multiobjective Optimization with Two Objectives".

If you export results using **File > Export to Workspace**, the exported results do not depend on the sorted display.

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 app.

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 app when you change solvers, select **File** > **Preferences**, which opens the Preferences dialog box shown below.
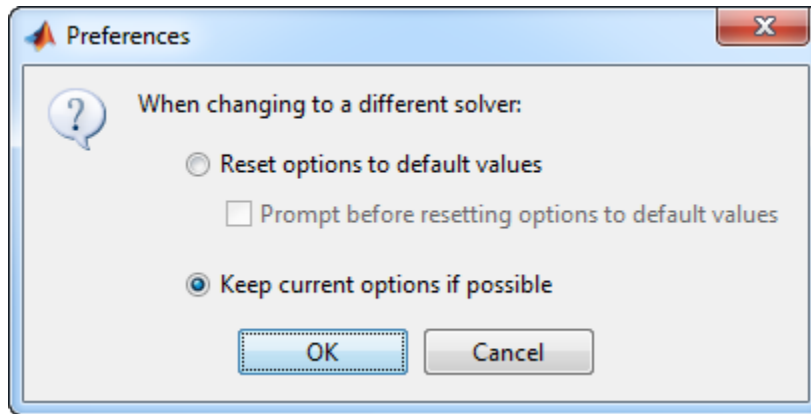
The default value, **Reset options to defaults**, discards any options you specified previously in the `optimtool`. Under this choice, you can select the option **Prompt before resetting options to defaults**.

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.

Specifying Certain Options

- "Plot Functions" on page 5-8
- "Output function" on page 5-9
- "Display to Command Window" on page 5-10

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 app, 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}
  ```
  Write custom plot functions with the same syntax as output functions. For information, see "Output Function" on page 13-20.
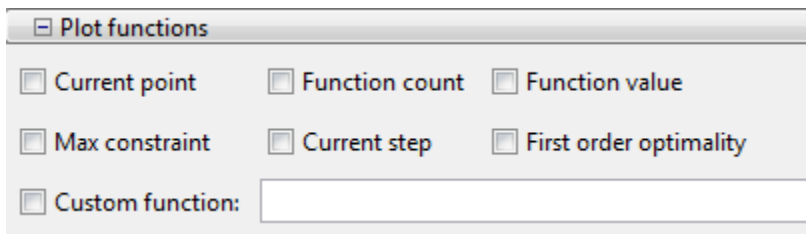
The graphic above shows the plot functions available for the default `fmincon` solver.

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 13-20.

```
□ Output function
□ Custom function:
```
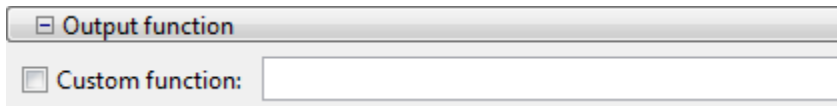
Display to Command Window

Select **Level of display** to specify the amount of information displayed when you run the algorithm. Choose from the following; depending on the solver, only some may be available:

- `off` (default) — Display no output.
- `final` — Display the reason for stopping at the end of the run.
- `final with detailed message` — Display the detailed reason for stopping at the end of the run.
- `notify` — Display output only if the function does not converge.
- `notify with detailed message` — Display a detailed output only if the function does not converge.
- `iterative` — Display information at each iteration of the algorithm and the reason for stopping at the end of the run.
- `iterative with detailed message` — Display information at each iteration of the algorithm and the detailed reason for stopping at the end of the run.

See "Enhanced Exit Messages" on page 3-5 for information on detailed messages.

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

The graphic below shows the display options for the `fmincon` solver. Some other solvers have fewer options.
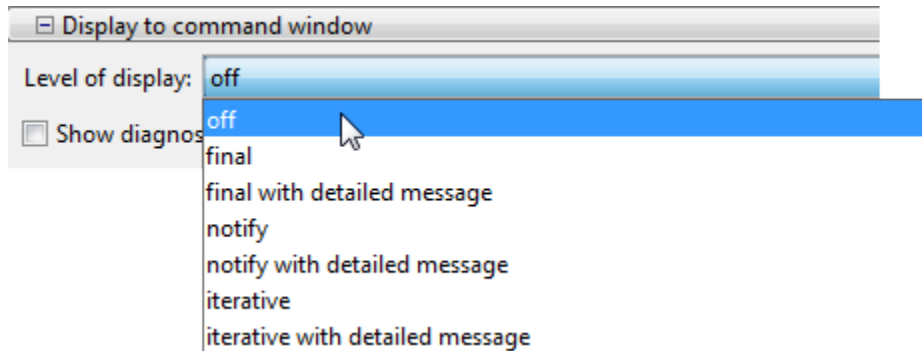
Importing and Exporting Your Work

Exporting Your Work

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

To access the **Export to Workspace** dialog box shown below, select **File > Export to Workspace**.
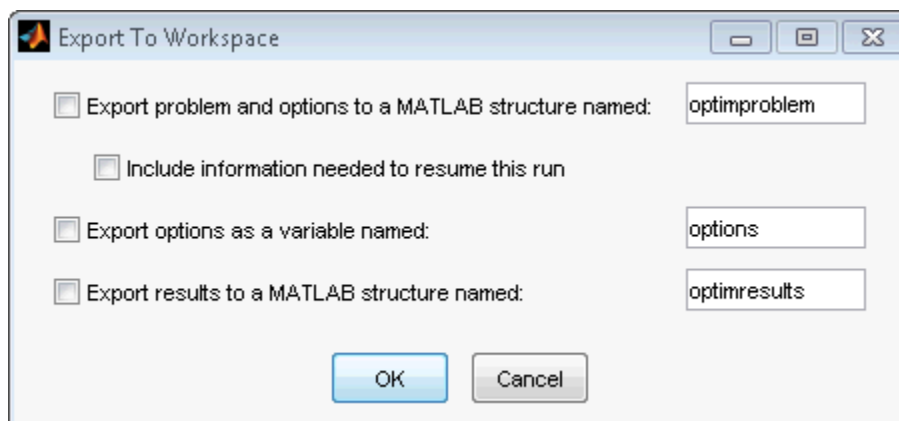
You can specify results that contain:

- 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 the Simulated Annealing solver)
- 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 the Optimization app

Exported results 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 app 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 or object, double-click the name in the Workspace window. Alternatively, see the values by entering `exportname.fieldname` at the command line. For example, so see the message in an output structure, enter `output.message`. If a structure contains structures or objects, you can double-click again in the workspace browser, or enter `exportname.name2.fieldname` at the command line. For example, to see the level of iterative display contained in the options 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 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 save options from Optimization Toolbox functions at the Command Window, or whether you export options, or the problem and options, from the Optimization app, you can resume work on your problem using the Optimization app.

There are three ways to import your options, or problem and options, to the Optimization app:

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

  `optimtool(options)`
- Select **File** > **Import Options** in the Optimization app.
- Select **File** > **Import Problem** in the Optimization app.

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 the Simulated Annealing solver, 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 a File

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

To export data from the Optimization app to a file, select **File** > **Generate Code**.

The generated 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 with the currently selected option value

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

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

6

Nonlinear algorithms and examples

Unconstrained Nonlinear Optimization Algorithms

In this section...

Unconstrained Optimization 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.

fminunc `trust-region` 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,

$$\min_s \{q(s),\ s \in N\}.$$

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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \ \text{ such that } \ \|Ds\| \leq \Delta \right\},$$

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 6-2 (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 6-2. However, they require time proportional to several factorizations of H. Therefore, for large-scale problems a different approach is needed. Several approximation and heuristic strategies, based on Equation 6-2, 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 6-2 is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The 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

$$H \cdot s_2 = -g,$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$

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 6-2 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., $d^T H d \leq 0$. 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 6-2.

`fminunc quasi-newton` 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

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

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.
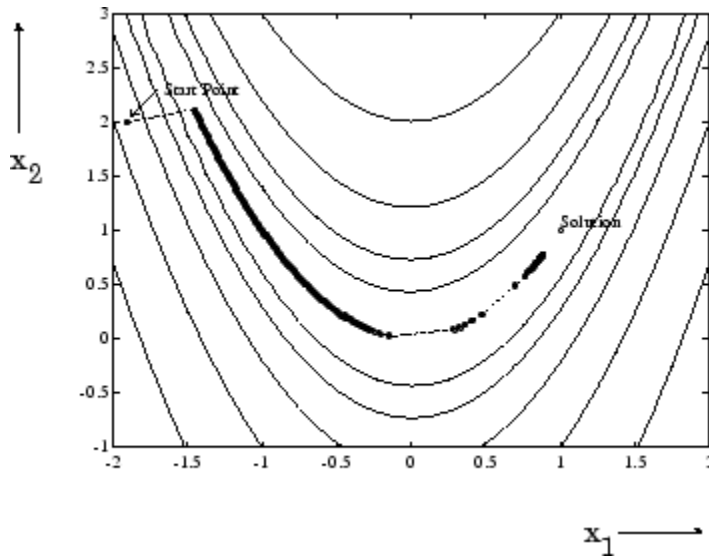
Figure 6-1. Steepest Descent Method on Rosenbrock's Function

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.

For an animated version of this figure, enter `bandem` at the MATLAB command line.

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

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

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.,

$$\nabla f(x^*) = Hx^* + c = 0.$$

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

$$x^* = -H^{-1}c.$$

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

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

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 6-9) 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

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

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function in Figure 6-2, 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.
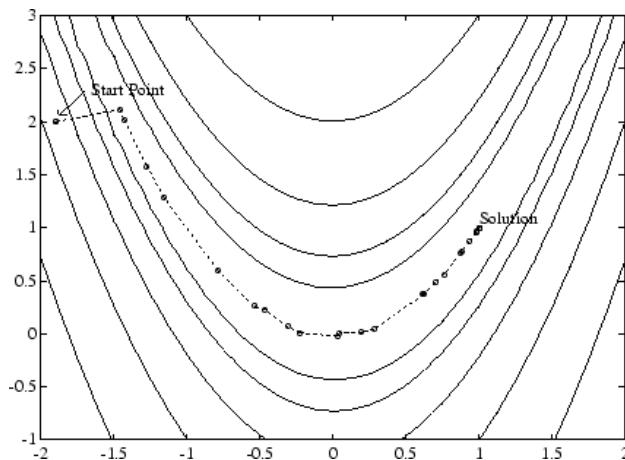
Figure 6-2. BFGS Method on Rosenbrock's Function

For an animated version of this figure, enter `bandem` at the MATLAB command line.

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

$$x_{k+1} = x_k + \alpha^* d_k,$$

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:

$$f\left(x_k + \alpha d_k\right) \leq f\left(x_k\right) + c_1 \alpha \nabla f_k^T d_k,$$

$$\nabla f\left(x_k + \alpha d_k\right)^T d_k \geq c_2 \nabla f_k^T d_k,$$

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

The first condition (Equation 6-12) requires that α_k sufficiently decreases the objective function. The second condition (Equation 6-13) ensures that the step length is not too small. Points that satisfy both conditions (Equation 6-12 and Equation 6-13) 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 6-9). The

function `fminunc` also provides an option to use the DFP method given in "Quasi-Newton Methods" on page 6-6 (set `HessUpdate` to `'dfp'` in `options` to select the DFP method). The Hessian, H, is always maintained to be positive definite so that the direction of search, d, is always in a descent direction. This means that for some arbitrarily small step a 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 6-14) is always positive. The term $q_k^T s_k$ is a product of the line search step length parameter a_k and a combination of the search direction d with past and present gradient evaluations,

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

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 a_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 Lagarias et al. [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 keywords for the `fminsearch` iterative display appear in bold after the description of the step.

---

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 algorithm discards the current worst point $x(n+1)$,

and accepts another point into the simplex. [Or, in the case of step 7 below, it changes all *n* points with values above $f(x(1))$].

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 $f(x(1)) \leq f(r) < f(x(n))$, 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 $f(r) \geq f(x(n))$, 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 $f(r) \geq f(x(n+1))$, 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**

The following figure shows the points that `fminsearch` might calculate in the procedure, along with each possible new simplex. The original simplex has a bold outline. The iterations proceed until they meet a stopping criterion.
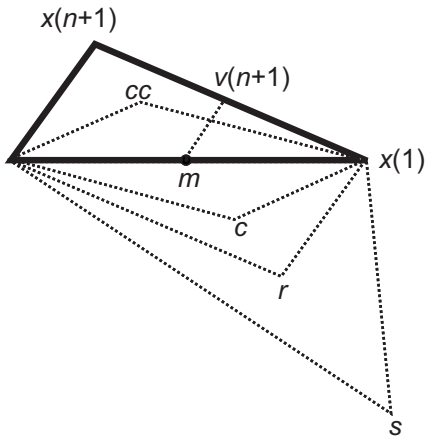
fminunc Unconstrained Minimization

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

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

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

Step 1: Write a file objfun.m.

This code ships with the toolbox. To view, enter type objfun:

```
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: Set options.

Set options to use the 'quasi-newton' algorithm. Set options because the 'trust-region' algorithm requires that the objective function include a gradient. If you do not set the options, then, depending on your MATLAB version, fminunc can issue a warning.

```
options = optimoptions(@fminunc,'Algorithm','quasi-newton');
```

Step 3: Invoke fminunc using the options.

```
x0 = [-1,1]; % Starting guess
[x,fval,exitflag,output] = fminunc(@objfun,x0,options);
```

This produces the following output:

```
Local minimum found.

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

View the results:

```
x,fval,exitflag,output
```

```
x =
    0.5000   -1.0000

fval =
  3.6609e-15

exitflag =
     1

output =
   iterations: 8
    funcCount: 66
     stepsize: 6.3361e-07
 lssteplength: 1
firstorderopt: 1.2284e-07
    algorithm: 'quasi-newton'
      message: 'Local minimum found.…'
```

The `exitflag` tells whether the algorithm converged. `exitflag = 1` means a local minimum was found. The meanings of exitflags 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).

Related Examples
- "Minimization with Gradient and Hessian" on page 6-15

More About
- "Set Options"
- "Solver Outputs and Iterative Display"

Minimization with Gradient and Hessian

This example shows how to solve a nonlinear minimization problem with an explicit tridiagonal Hessian matrix $H(x)$.

The problem is to find x to minimize

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

where $n = 1000$.

Step 1: Write a 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 `optimoptions` 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 = optimoptions(@fminunc,'Algorithm','trust-region',...
    '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:

```
fval,exitflag,output
```

```
fval =
   2.8709e-17

exitflag =
     1

output =
         iterations: 7
          funcCount: 8
           stepsize: 0.0039
       cgiterations: 7
      firstorderopt: 4.7948e-10
          algorithm: 'trust-region'
            message: 'Local minimum found.…'
    constrviolation: []
```

Related Examples

- "Minimization with Gradient and Hessian Sparsity Pattern" on page 6-17

Minimization with Gradient and Hessian Sparsity Pattern

This example shows how to solve a nonlinear minimization problem with tridiagonal Hessian matrix approximated by sparse finite differences instead of explicit computation.

The problem is to find x to minimize

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

where $n = 1000$.

To use the `trust-region` method in `fminunc`, you *must* compute the gradient in `fun`; it is *not* optional as in the `quasi-newton` method.

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

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

This function file ships with your software.

```
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 optimoptions 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 optimoptions, 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
```
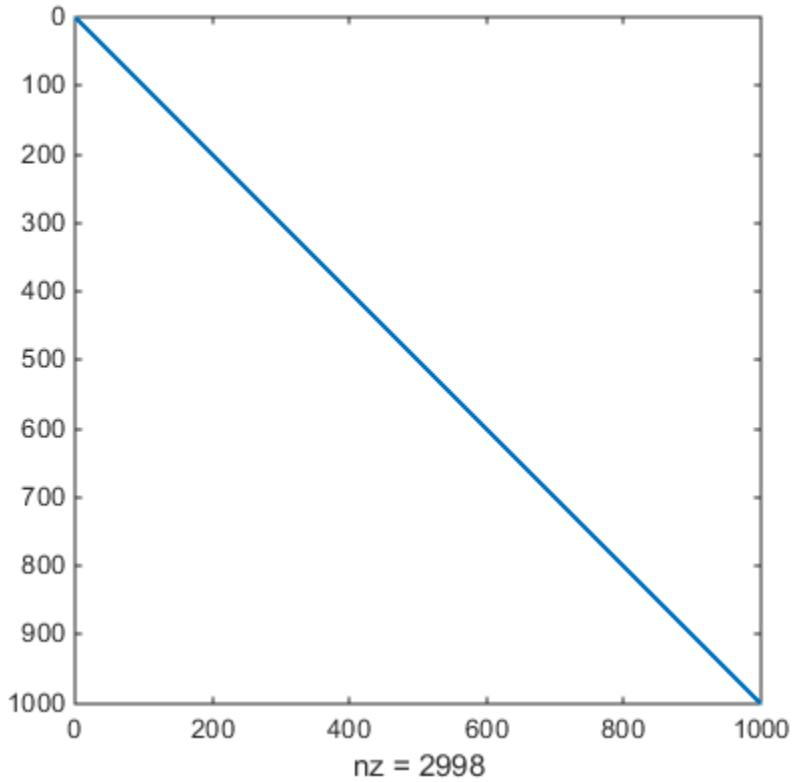
nz = 2998

```
n = 1000;
xstart = -ones(n,1);
xstart(2:2:n,1) = 1;
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'GradObj','on','HessPattern',Hstr);
[x,fval,exitflag,output] = fminunc(fun,xstart,options);
```

This 1000-variable problem is solved in seven 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,fval,output
```

```
exitflag =
     1

fval =
   7.4738e-17

output =
          iterations: 7
           funcCount: 8
            stepsize: 0.0046
         cgiterations: 7
       firstorderopt: 7.9822e-10
           algorithm: 'trust-region'
             message: 'Local minimum found.…'
      constrviolation: []
```

Related Examples

*   "Minimization with Gradient and Hessian" on page 6-15

Constrained Nonlinear Optimization Algorithms

In this section...
"Constrained Optimization Definition" on page 6-21
"fmincon Trust Region Reflective Algorithm" on page 6-21
"fmincon Active Set Algorithm" on page 6-26
"fmincon SQP Algorithm" on page 6-36
"fmincon Interior Point Algorithm" on page 6-37
"fminbnd Algorithm" on page 6-41
"fseminf Problem Formulation and Algorithm" on page 6-41

Constrained Optimization 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: $c(x) \le 0$, $ceq(x) = 0$, $A\,x \le b$, $Aeq\,x = beq$, $l \le x \le u$. There are even more constraints used in semi-infinite programming; see "fseminf Problem Formulation and Algorithm" on page 6-41.

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,

$$\min_s \{q(s),\ s \in N\}.$$

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

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

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 6-18 (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 6-18. 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 6-18, 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 6-18 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

$$H \cdot s_2 = -g,$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$

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 6-18 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., $d^THd \leq 0$. 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 6-2.

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 trust-region methods in Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\},$$

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 6-21 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

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

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

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

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 6-23,

$$(D(x))^{-2} g = 0,$$

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 6-24 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 6-24 is defined as the solution to the linear system

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

at the kth iteration, where

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

and

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

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. $J^v_{ii} = 0$ is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value v_i takes. Further, $|v_i|$ will still be discontinuous at this point, but the function $|v_i| \cdot g_i$ is continuous.

Second, reflections are used to increase the 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 2-1), the Kuhn-Tucker equations can be stated as

$$\nabla f\left(x^*\right) + \sum_{i=1}^{m} \lambda_i \cdot \nabla G_i\left(x^*\right) = 0$$

$$\lambda_i \cdot G_i\left(x^*\right) = 0, \quad i = 1,...,m_e$$

$$\lambda_i \geq 0, \quad i = m_e + 1,...,m,$$

in addition to the original constraints in Equation 2-1.

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 2-12.

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 2-1) the principal idea is the formulation of a QP subproblem based on a quadratic approximation of the Lagrangian function.

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

Here you simplify Equation 2-1 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

$$\min_{d \in \Re^n} \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d$$

$$\nabla g_i(x_k)^T d + g_i(x_k) = 0, \quad i = 1,\dots,m_e$$

$$\nabla g_i(x_k)^T d + g_i(x_k) \le 0, \quad i = m_e + 1,\dots,m.$$

This subproblem can be solved using any QP algorithm (see, for instance, "Quadratic Programming Solution" on page 6-31). 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 6-30). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function (Equation 6-29). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see "Updating the Hessian Matrix" on page 6-30) appears to be the most popular.

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

Consider Rosenbrock's function with an additional nonlinear inequality constraint, $g(x)$,

$$x_1^2 + x_2^2 - 1.5 \le 0.$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. SQP Method on Nonlinearly Constrained Rosenbrock's Function shows the path to the solution point $x = [0.9072, 0.8228]$ starting at $x = [-1.9, 2.0]$.
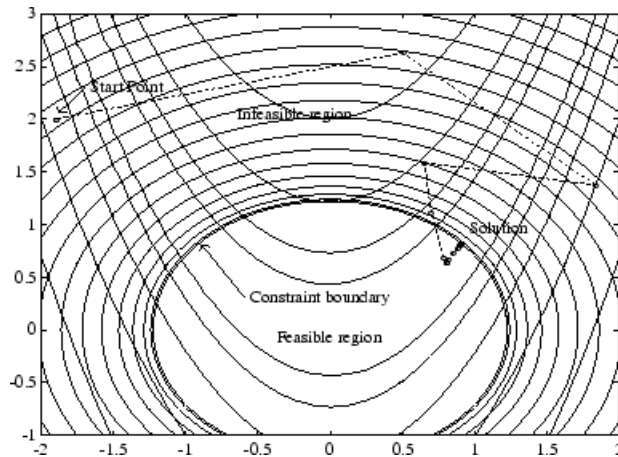
Figure 6-3. SQP Method on Nonlinearly Constrained Rosenbrock's Function

SQP Implementation

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

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.

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

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,

$$q_k = q_k + wv,$$

where

$$v_i = \nabla g_i \left(x_{k+1} \right) \cdot g_i \left(x_{k+1} \right) - \nabla g_i \left(x_k \right) \cdot g_i \left(x_k \right)$$
$$\text{if } \left(q_k \right)_i \cdot w < 0 \text{ and } \left(q_k \right)_i \cdot \left(s_k \right)_i < 0, \ i = 1, \ldots, m$$
$$v_i = 0 \ \text{otherwise,}$$

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 `ith` row of the m-by-n matrix A.

$$\min_{d \in \mathfrak{R}^n} q(d) = \frac{1}{2} d^T H d + c^T d,$$
$$A_i d = b_i, \ \ i = 1, \ldots, m_e$$
$$A_i d \leq b_i, \ \ i = m_e + 1, \ldots, m.$$

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,

$\overline{A}_k$, 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

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

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

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

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c.$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c.$$

A step is then taken of the form

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

At each iteration, because of the quadratic nature of the objective function, there are only two choices of step length a. 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 6-34). 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

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

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

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

$$\bar{A}_k^T \lambda_k = c.$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 6-34). 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

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

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 6-42 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).

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

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

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

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

$$x_{k+1} = x_k + \alpha d_k.$$

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.

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

Powell recommends setting the penalty parameter

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

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

$$r_i = \frac{\|\nabla f(x)\|}{\|\nabla g_i(x)\|},$$

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 SQP Algorithm

The `sqp` algorithm is similar to the `active-set` algorithm (for a description, see "fmincon Active Set Algorithm" on page 6-26). The basic `sqp` algorithm is described in Chapter 18 of Nocedal and Wright [31].

The most important differences between the `sqp` and the `active-set` algorithms are:

Strict Feasibility With Respect to Bounds

The `sqp` algorithm takes every iterative step in the region constrained by bounds. Furthermore, finite difference steps also respect bounds. Bounds are not strict; a step can be exactly on a boundary. This strict feasibility can be beneficial when your objective function or nonlinear constraint functions are undefined or are complex outside the region constrained by bounds.

Robustness to Non-Double Results

During its iterations, the `sqp` algorithm can attempt to take a step that fails. This means an objective function or nonlinear constraint function you supply returns a value of `Inf`, `NaN`, or a complex value. In this case, the algorithm attempts to take a smaller step.

Refactored Linear Algebra Routines

The `sqp` algorithm uses a different set of linear algebra routines to solve the quadratic programming subproblem, Equation 6-30. These routines are more efficient in both memory usage and speed than the `active-set` routines.

Reformulated Feasibility Routines

The `sqp` algorithm has two new approaches to the solution of Equation 6-30 when constraints are not satisfied.

- The `sqp` algorithm combines the objective and constraint functions into a merit function. The algorithm attempts to minimize the merit function subject to relaxed constraints. This modified problem can lead to a feasible solution. However, this approach has more variables than the original problem, so the problem size in Equation 6-30 increases. The increased size can slow the solution of the subproblem.

These routines are based on the articles by Spellucci [60] and Tone [61]. The `sqp` algorithm sets the penalty parameter for the merit function Equation 6-46 according to the suggestion in [41].

- Suppose nonlinear constraints are not satisfied, and an attempted step causes the constraint violation to grow. The `sqp` algorithm attempts to obtain feasibility using a second-order approximation to the constraints. The second-order technique can lead to a feasible solution. However, this technique can slow the solution by requiring more evaluations of the nonlinear constraint functions.

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

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

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

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

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 6-50 is a sequence of equality constrained problems. These are easier to solve than the original inequality-constrained problem Equation 6-49.

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 3-2 and Equation 3-3, 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

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

The parameter v 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.

If either the objective function or a nonlinear constraint function returns a complex value, NaN, Inf, or an error at an iterate x_j, the algorithm rejects x_j. The rejection has the same effect as if the merit function did not decrease sufficiently: the algorithm then attempts a different, shorter step. Wrap any code that can error in `try-catch`:

```
function val = userFcn(x)
try
    val = ... % code that can error
catch
    val = NaN;
end
```

The objective and constraints must yield proper (`double`) values at the initial point.

Direct Step

The following variables are used in defining the direct step:

- H denotes the Hessian of the Lagrangian of f_μ:

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j \lambda_j \nabla^2 h_j(x).$$

- J_g denotes the Jacobian of the constraint function g.
- J_h denotes the Jacobian of the constraint function h.
- $S = \text{diag}(s)$.
- λ denotes the Lagrange multiplier vector associated with constraints g
- $\Lambda = \text{diag}(\lambda)$.

- y denotes the Lagrange multiplier vector associated with h.
- e denote the vector of ones the same size as g.

Equation 6-52 defines the direct step $(\Delta x, \Delta s)$:

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

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

In order to solve this equation for $(\Delta x, \Delta s)$, the algorithm makes an LDL factorization of the matrix. (See Example 3 — 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 6-50 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 6-38. 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 $(\Delta x, \Delta s)$ to approximately solve

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

subject to the linearized constraints

$$g(x) + J_g \Delta x + \Delta s = 0, \ \ h(x) + J_h \Delta x = 0.$$

To solve Equation 6-54, the algorithm tries to minimize a norm of the linearized constraints inside a region with radius scaled by R. Then Equation 6-53 is solved with the constraints being to match the residual from solving Equation 6-54, 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 6-23.

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 14-60 and "Hessian Multiply Function" on page 14-62 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** in fmincon options.

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 $c(x) \leq 0$, $ceq(x) = 0$, $A\,x \leq b$, $Aeq\,x = beq$, and $l \leq x \leq u$.

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
$K_j(x, w_j) \leq 0$ for all $w_j \in I_j$.

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: $K_j(x, w_j) \leq 0$ 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

$$\max_{w_j \in I_j} K_j(x, w_j) \le 0 \text{ for all } j = 1, ..., |K|,$$

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 6-55. 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 $c(x) \leq 0$, $ceq(x) = 0$, $A\ x \leq b$, $Aeq\ x = beq$, and $l \leq x \leq u$, where $c(x)$ is augmented with all the maxima of $\kappa_j(x,\ w_j)$ taken over all $w_j \in I_j$, 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.

Nonlinear Inequality Constraints

This example shows how to solve a scalar minimization problem with nonlinear inequality constraints. The problem is to find x that solves

$$\min_{x} f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1 x_2 + 2x_2 + 1\right).$$

subject to the constraints
$x_1 x_2 - x_1 - x_2 \leq -1.5,$
$x_1 x_2 \geq -10.$

Because neither of the constraints is linear, you cannot pass the constraints to `fmincon` at the command line. Instead you can create a second 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

$x_1 x_2 - x_1 - x_2 + 1.5 \leq 0,$
$-x_1 x_2 - 10 \leq 0.$

Step 1: Write a 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 a 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 = optimoptions(@fmincon,'Algorithm','sqp');
[x,fval] = ...
```

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

`fmincon` produces the solution x with function value `fval`:

```
x,fval
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-14 *

   -0.6661
    0.7105


ceq =

     []
```

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

Related Examples

- "Nonlinear Equality and Inequality Constraints" on page 6-61
- "Nonlinear Constraints with Gradients" on page 6-47

Nonlinear Constraints with Gradients

Ordinarily, 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.

Consider how to solve

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

subject to the constraints
$x_1 x_2 - x_1 - x_2 \leq -1.5,$
$x_1 x_2 \geq -10.$

To solve the problem using analytically determined gradients, do the following.

Step 1: Write a file for the objective function and gradient.

```
function [f,gradf] = 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
    gradf = [ f + exp(x(1)) * (8*x(1) + 4*x(2)),
    exp(x(1))*(4*x(1)+4*x(2)+2)];
end
```

Step 2: Write a 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
```

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

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

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

$$\begin{bmatrix} \dfrac{\partial c_1}{\partial x_1} & \dfrac{\partial c_2}{\partial x_1} \\ \dfrac{\partial c_1}{\partial x_2} & \dfrac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}.$$

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 files contain this additional information. Use `optimoptions` to turn the options `GradObj` and `GradConstr` to `'on'` in the example's existing `options`:

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

If you do not set these options to `'on'`, `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, so set both to `[]`.

Step 3: Invoke the constrained optimization routine.

```
x0 = [-1,1];            % Starting guess
options = optimoptions(@fmincon,'Algorithm','sqp');
options = optimoptions(options,'GradObj','on','GradConstr','on');
lb = [ ]; ub = [ ];   % No upper or lower bounds
[x,fval] = fmincon(@objfungrad,x0,[],[],[],[],lb,ub,...
    @confungrad,options);
```

The results:

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

[c,ceq] = confungrad(x) % Check the constraint values at x
c =
   1.0e-13 *
   -0.1066
    0.1066

ceq =
     []
```

fmincon Interior-Point Algorithm with Analytic Hessian

The `fmincon` 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]:
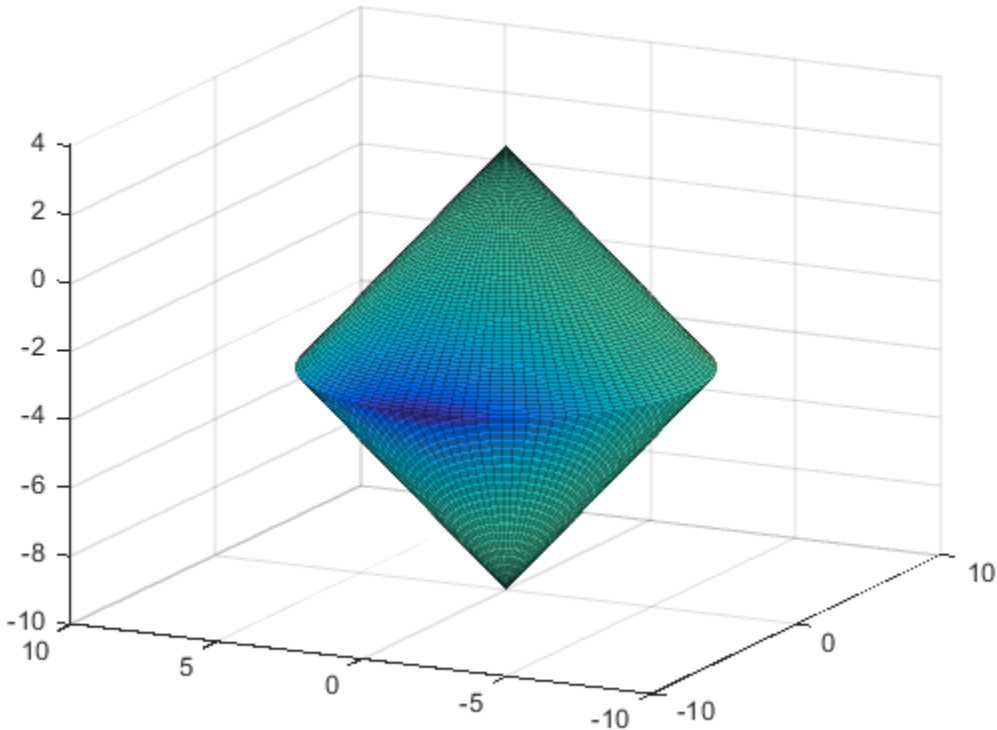
The Hessian of the Lagrangian is given by the equation:

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

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 app:

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 = optimoptions(@fmincon,'Algorithm','interior-point',...
                'Display','off','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:

```
x =
   -6.5000   -0.0000   -3.5000

fval =
  -2.8941e+03

mflag =
     1

output =
          iterations: 6
           funcCount: 7
       constrviolation: 0
            stepsize: 3.0479e-05
           algorithm: 'interior-point'
        firstorderopt: 5.9812e-05
         cgiterations: 3
             message: 'Local minimum found that satisfies the constraints.…'
```

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

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
        'Display','off','GradObj','on','GradConstr','on');

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

x =
   -6.5000   -0.0000   -3.5000

fval =
  -2.8941e+03

mflag =
```

```
     1

output =
         iterations: 9
          funcCount: 13
     constrviolation: 2.9391e-08
            stepsize: 6.4842e-04
           algorithm: 'interior-point'
       firstorderopt: 1.4235e-04
         cgiterations: 0
             message: 'Local minimum found that satisfies the constraints.…'
```

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

Related Examples

- "Linear or Quadratic Objective with Quadratic Constraints" on page 6-56
- "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80

Linear or Quadratic Objective with Quadratic Constraints

This example shows how to solve an optimization problem that has a linear or quadratic objective and quadratic inequality constraints. It shows how to generate and use the gradient and Hessian of the objective and constraint functions.

Quadratic Constrained Problem

Suppose that you can put your problem in the form

$$\min_{x} \left(\frac{1}{2} x^T Q x + f^T x + c \right)$$

subject to

$$\frac{1}{2} x^T H_i x + k_i^T x + d_i \leq 0,$$

where $1 \leq i \leq m$. Assume that at least one H_i is nonzero; otherwise, you can use quadprog or linprog to solve this problem. With nonzero H_i, the constraints are nonlinear, and the "Optimization Decision Table" on page 2-5 states that fmincon is the appropriate solver.

If x has N components, then Q and the H_i are N-by-N matrices, f and the k_i are N-by-1 vectors, and c and the d_i are scalars.

Objective Function

Formulate the problem using fmincon syntax. Assume that x and f are column vectors. (x is a column vector when the initial vector x0 is.)

```
function [y,grady] = quadobj(x,Q,f,c)
y = 1/2*x'*Q*x + f'*x + c;
if nargout > 1
    grady = Q*x + f;
end
```

Constraint Function

For consistency and easy indexing, place every quadratic constraint matrix in one cell array. Similarly, place the linear and constant terms in cell arrays.

```
function [y,yeq,grady,gradyeq] = quadconstr(x,H,k,d)
jj = length(H); % jj is the number of inequality constraints
y = zeros(1,jj);
for i = 1:jj
    y(i) = 1/2*x'*H{i}*x + k{i}'*x + d{i};
end
yeq = [];

if nargout > 2
    grady = zeros(length(x),jj);
    for i = 1:jj
        grady(:,i) = H{i}*x + k{i};
    end
end
gradyeq = [];
```

Numeric Example

For example, suppose that you have the following problem.

```
Q = [3,2,1;
     2,4,0;
     1,0,5];
f = [-24;-48;-130];
c = -2;

rng default % for reproducibility
% Two sets of random quadratic constraints:
H{1} = gallery('randcorr',3); % random positive definite matrix
H{2} = gallery('randcorr',3);
k{1} = randn(3,1);
k{2} = randn(3,1);
d{1} = randn;
d{2} = randn;
```

Hessian

Create a Hessian function. The Hessian of the Lagrangian is given by the equation

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

fmincon calculates an approximate set of Lagrange multipliers λ_i, and packages them in a structure. To include the Hessian, use the following function.

```
function hess = quadhess(x,lambda,Q,H)
hess = Q;
jj = length(H); % jj is the number of inequality constraints
for i = 1:jj
    hess = hess + lambda.ineqnonlin(i)*H{i};
end
```

Solution

Use the `fmincon` `interior-point` algorithm to solve the problem most efficiently. This algorithm accepts a Hessian function that you supply. Set these options.

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
    'GradObj','on','GradConstr','on','Hessian','user-supplied',...
    'HessFcn',@(x,lambda)quadhess(x,lambda,Q,H));
```

Call `fmincon` to solve the problem.

```
fun = @(x)quadobj(x,Q,f,c);
nonlconstr = @(x)quadconstr(x,H,k,d);
x0 = [0;0;0]; % column vector
[x,fval,eflag,output,lambda] = fmincon(fun,x0,...
    [],[],[],[],[],[],nonlconstr,options);
```

Examine the Lagrange multipliers.

```
lambda.ineqnonlin
```

```
ans =

    12.8412
    39.2337
```

Both nonlinear inequality multipliers are nonzero, so both quadratic constraints are active at the solution.

Efficiency When Providing a Hessian

The interior-point algorithm with gradients and a Hessian is efficient. Examine the number of function evaluations.

```
output
```

```
output =
```

```
          iterations: 9
           funcCount: 10
       constrviolation: 0
            stepsize: 5.3547e-04
           algorithm: 'interior-point'
        firstorderopt: 1.5851e-05
         cgiterations: 0
             message: 'Local minimum found that satisfies the constraints.
```

Optimization compl...'

`fmincon` used just 10 function evaluations to solve the problem.

Compare this to the solution without the Hessian.

```
options.Hessian = 'off';
[x2,fval2,eflag2,output2,lambda2] = fmincon(fun,[0;0;0],...
    [],[],[],[],[],[],nonlconstr,options);
output2
```

output2 =

```
          iterations: 17
           funcCount: 22
       constrviolation: 0
            stepsize: 2.8475e-04
           algorithm: 'interior-point'
        firstorderopt: 1.7680e-05
         cgiterations: 0
             message: 'Local minimum found that satisfies the constraints.
```

Optimization compl...'

This time `fmincon` used about twice as many iterations and function evaluations. The solutions are the same to within tolerances.

Extension to Quadratic Equality Constraints

If you also have quadratic equality constraints, you can use essentially the same technique. The problem is the same, with the additional constraints

$$\frac{1}{2}x^T J_i x + p_i^T x + q_i = 0.$$

Reformulate your constraints to use the J_i, p_i, and q_i variables. The `lambda.eqnonlin(i)` structure has the Lagrange multipliers for equality constraints.

Related Examples

- "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50

More About

- "Including Derivatives" on page 2-20
- "Including Gradients in Constraint Functions" on page 2-35

Nonlinear Equality and Inequality Constraints

You can include nonlinear constraints by writing a function that computes both equality and inequality constraint values. A nonlinear constraint function has the syntax

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

The function `c(x)` represents the constraint `c(x) <= 0`. The function `ceq(x)` represents the constraint `ceq(x) = 0`.

Note: You must have the nonlinear constraint function return both `c(x)` and `ceq(x)`, even if you have only one type of nonlinear constraint. If a constraint does not exist, have the function return `[ ]` for that constraint.

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 a 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 a 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 = optimoptions(@fmincon,'Algorithm','sqp');
[x,fval] = fmincon(@objfun,x0,[],[],[],[],[],[],...
   @confuneq,options);
```

After 21 function evaluations, the solution produced is

```
x,fval
x =
   -0.7529    0.4332
fval =
    1.5093

[c,ceq] = confuneq(x) % Check the constraint values at x

c =
   -9.6739
ceq =
   5.3291e-15
```

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.

Related Examples

- "Nonlinear Inequality Constraints" on page 6-45
- "Optimization App with the fmincon Solver" on page 6-63

Optimization App with the fmincon Solver

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

Note: The Optimization app warns that it will be removed in a future release.

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

$$
\begin{aligned}
0.5 &\leq x_1 && \text{(bound)} \\
-x_1 - x_2 + 1 &\leq 0 && \text{(linear inequality)}
\end{aligned}
$$

$$
\left.
\begin{aligned}
-x_1^2 - x_2^2 + 1 &\leq 0 \\
-9x_1^2 - x_2^2 + 9 &\leq 0 \\
-x_1^2 + x_2 &\leq 0 \\
-x_2^2 + x_1 &\leq 0
\end{aligned}
\right\} \quad \text{(nonlinear inequality)}
$$

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

Step 1: Write a file objecfun.m for the objective function.

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

Step 2: Write a file nonlconstr.m for the nonlinear 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 app.

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

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

Solver:	fmincon - Constrained nonlinear minimization	▼
Algorithm:	Active set	▼

3   Enter `@objecfun` in the **Objective function** field to call the `objecfun.m` file.

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

Objective function:	@objecfun	▼
Derivatives:	Approximated by solver	▼
Start point:	[3;1]	

5   Define the constraints.

- Set the bound $0.5 \leq x_1$ by entering `[0.5,-Inf]` in the **Lower** field. The `-Inf` entry means there is no lower bound on x_2.

- Set the linear inequality constraint by entering `[-1  -1]` in the **A** field and enter `-1` in the **b** field.

- Set the nonlinear constraints by entering `@nonlconstr` in the **Nonlinear constraint function** field.

Constraints:				
Linear inequalities:	A:	[-1 -1]	b:	-1
Linear equalities:	Aeq:		beq:	
Bounds:	Lower:	[0.5,-Inf]	Upper:	
Nonlinear constraint function:	@nonlconstr			
Derivatives:	Approximated by solver	▼		

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.
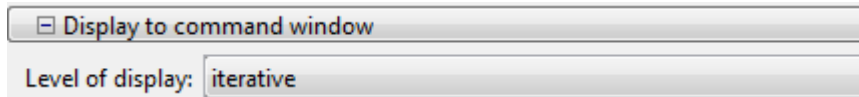
7   Click the **Start** button as shown in the following figure.
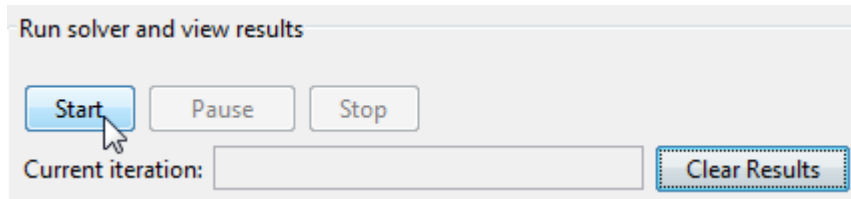
8   When the algorithm terminates, under **Run solver and view results** the following information is displayed:
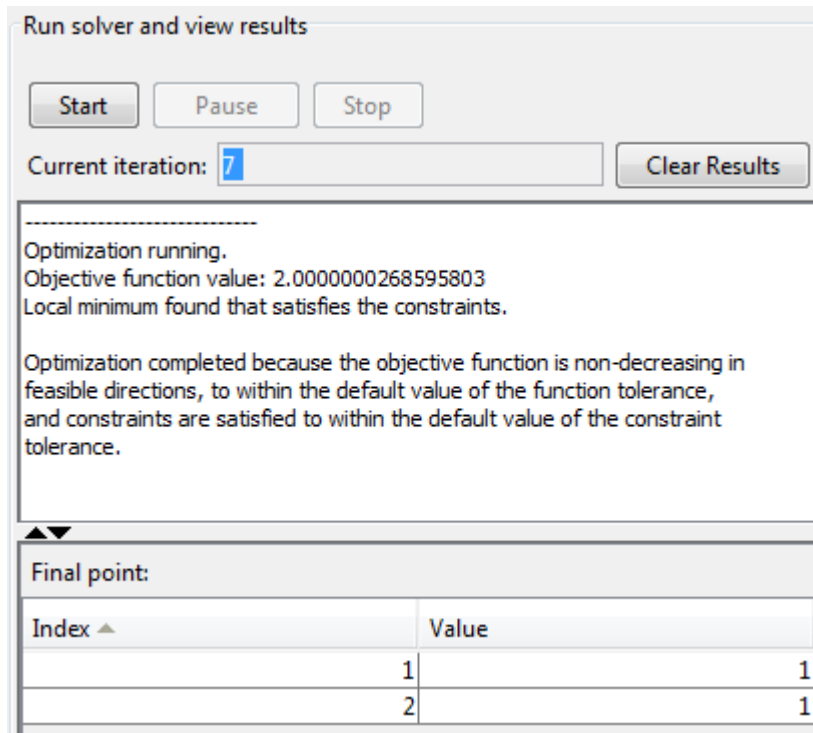
- The **Current iteration** value when the algorithm terminated, which for this example is 7.

- The final value of the objective function when the algorithm terminated:

  ```
  Objective function value: 2.0000000268595803
  ```

- The algorithm termination message:

  ```
  Local minimum found that satisfies the constraints.

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

- 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 Procedure
   0     3        10           2                                       Infeasible start point
   1     6   4.84298     -0.1322           1       -5.22        1.74
   2     9    4.0251     -0.01168          1       -4.39        4.08  Hessian modified twice
   3    12   2.42704     -0.03214          1       -3.85        1.09
   4    15   2.03615    -0.004728          1       -3.04       0.995  Hessian modified twice
   5    18   2.00033   -5.596e-005         1       -2.82      0.0664  Hessian modified twice
   6    21         2   -5.327e-009         1       -2.81    0.000522  Hessian modified twice

Local minimum found that satisfies the constraints.

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

Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin    ineqnonlin
                                      3
                                      4
```

Minimization with Bound Constraints and Banded Preconditioner

The goal in this problem is to minimize the nonlinear function

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

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

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

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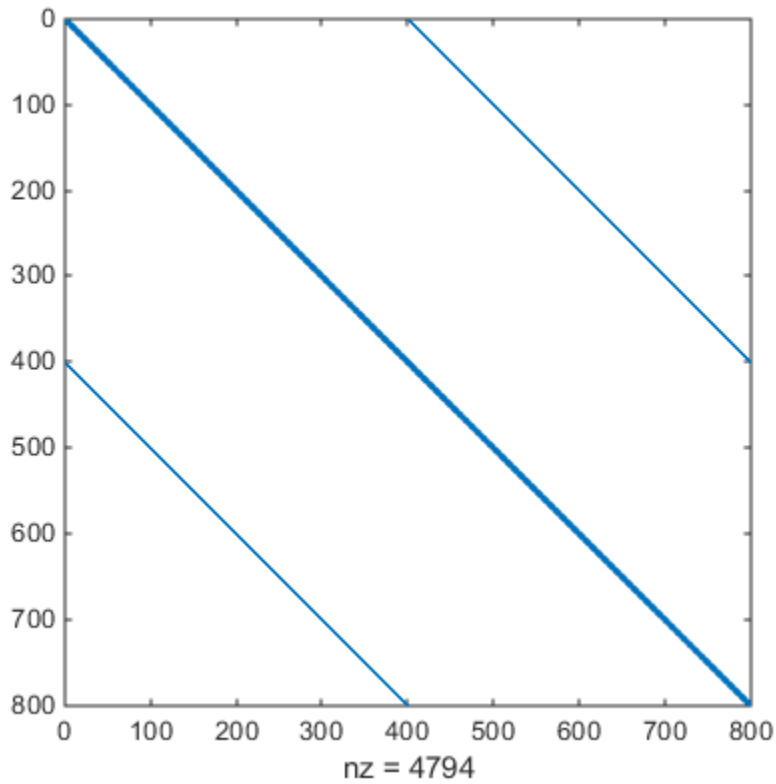
In this plot, the center stripe is itself a five-banded matrix. The following plot shows the matrix more clearly:

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

$nz = 94$
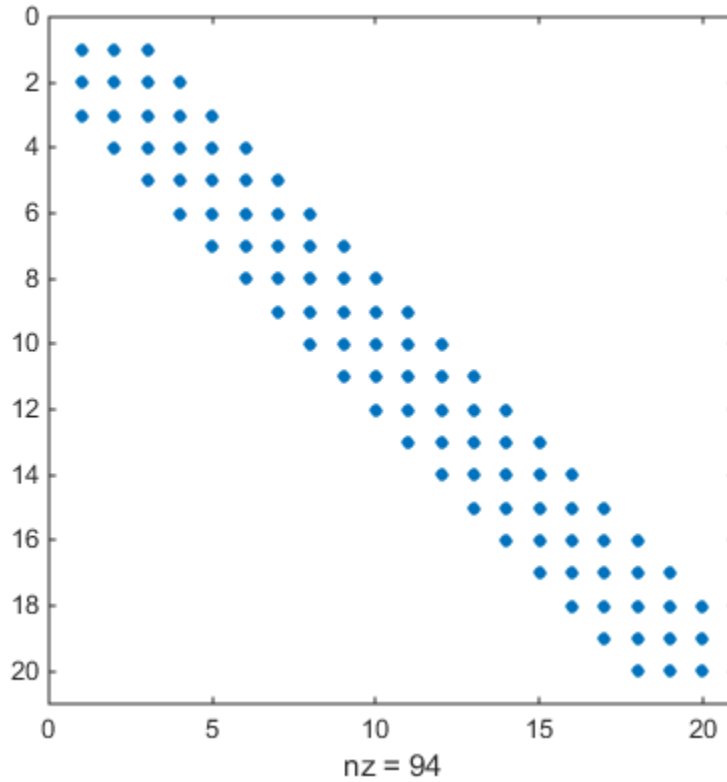
Use `optimoptions` to set the `HessPattern` parameter to `Hstr`. When a problem as large as this has obvious sparsity structure, not setting the `HessPattern` parameter requires a huge amount of unnecessary memory and computation. This is because `fmincon` attempts to use finite differencing on a full Hessian matrix of 640,000 nonzero entries.

You must also set the `GradObj` parameter to `'on'` using `optimoptions`, 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 = optimoptions('fmincon','GradObj','on','HessPattern',Hstr,...
    'Algorithm','trust-region-reflective');

[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
           stepsize: 8.2073e-04
       cgiterations: 18
      firstorderopt: 0.0163
          algorithm: 'trust-region-reflective'
            message: 'Local minimum possible.…'
     constrviolation: 0
```

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 6-25, 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 `optimoptions` 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 = optimoptions('fmincon','GradObj','on','HessPattern',Hstr, ...
    'Algorithm','trust-region-reflective','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
           stepsize: 2.4512e-05
       cgiterations: 15
      firstorderopt: 7.5340e-05
          algorithm: 'trust-region-reflective'
            message: 'Local minimum possible.…'
     constrviolation: 0
```

Minimization with Linear Equality Constraints

The trust-region reflective method for `fmincon` can handle linear equality constraints if no other constraints exist. Suppose you want to minimize

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

subject to some linear equality constraints. The objective function is coded in the function `brownfgh.m`. This example takes `n = 1000`. Furthermore, the `browneq.mat` file contains matrices `Aeq` and `beq` that represent the linear constraints *Aeq x = beq*. *Aeq* has 100 rows representing 100 linear constraints (so *Aeq* is a 100-by-1000 matrix).

Step 1: Write a 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 `optimoptions` 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 = optimoptions('fmincon','GradObj','on','Hessian','user-supplied',...
    'Algorithm','trust-region-reflective');
[x,fval,exitflag,output] = ...
    fmincon(fun,xstart,[],[],Aeq,beq,[],[],options);
```

`fmincon` prints the following exit message:

```
Local minimum possible.

fmincon stopped because the final change in function value relative to
its initial value is less than the default value of the function tolerance.
```

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

```
exitflag,fval,output

exitflag =
     3

fval =
  205.9313

output =
          iterations: 22
           funcCount: 23
            stepsize: 0.0054
         cgiterations: 30
        firstorderopt: 0.0027
           algorithm: 'trust-region-reflective'
             message: 'Local minimum possible.…'
     constrviolation: 2.2293e-13
```

The linear equalities are satisfied at `x`.

```
norm(Aeq*x-beq)

ans =
    1.1919e-12
```

Minimization with Dense Structured Hessian, Linear Equalities

In this section...
"Hessian Multiply Function for Lower Memory" on page 6-75
"Step 1: Write a file brownvv.m that computes the objective function, the gradient, and the sparse part of the Hessian." on page 6-76
"Step 2: Write a function to compute Hessian-matrix products for H given a matrix Y." on page 6-76
"Step 3: Call a nonlinear minimization routine with a starting point and linear equality constraints." on page 6-77
"Preconditioning" on page 6-79

Hessian Multiply Function for Lower Memory

The fmincon interior-point and trust-region-reflective algorithms, and the fminunc trust-region 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 trust-region algorithm is similar. For the interior-point algorithm, see the 'HessMult' option in "Hessian Multiply Function" on page 14-62. The objective function has the structure

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

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

$$H = \hat{H} - V V^T.$$

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 `optimoptions` 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 a 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 `optimoptions` 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` folder as `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 `optimdemos` folder. Use `optimoptions` 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.

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 = optimoptions(@fmincon,'Algorithm','trust-region-reflective','GradObj','on', ...
    'Hessian','user-supplied','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);
```

To run the preceding code, enter

```
[fval,exitflag,output,x] = runfleq1;
```

Because the iterative display was set using `optimoptions`, this command generates the following iterative display:

```
                          Norm of     First-order
Iteration       f(x)       step        optimality   CG-iterations
    0         2297.63                    1.41e+03
    1         1084.59      6.3903           578            1
    2         1084.59      100              578            3
    3         1084.59      25               578            0
    4         1084.59      6.25             578            0
    5         1047.61      1.5625           240            0
    6          761.592     3.125            62.4           2
    7          761.592     6.25             62.4           4
    8          746.478     1.5625           163            0
    9          546.578     3.125            84.1           2
   10          274.311     6.25             26.9           2
   11           55.6193    11.6597          40             2
   12           55.6193    25               40             3
   13           22.2964    6.25             26.3           0
   14          -49.516     6.25             78             1
   15          -93.2772    1.5625           68             1
   16         -207.204     3.125            86.5           1
   17         -434.162     6.25             70.7           1
   18         -681.359     6.25             43.7           2
   19         -681.359     6.25             43.7           4
   20         -698.041     1.5625           191            0
   21         -723.959     3.125            256            7
   22         -751.33      0.78125          154            3
   23         -793.974     1.5625           24.4           3
   24         -820.831     2.51937          6.11           3
   25         -823.069     0.562132         2.87           3
   26         -823.237     0.196753         0.486          3
   27         -823.245     0.0621202        0.386          3
   28         -823.246     0.0199951        0.11           6
   29         -823.246     0.00731333       0.0404         7
   30         -823.246     0.00505883       0.0185         8
   31         -823.246     0.00126471       0.00268        9
   32         -823.246     0.00149326       0.00521        9
   33         -823.246     0.000373314      0.00091        9
```

```
Local minimum possible.

fmincon stopped because the final change in function value relative to
its initial value is less than the selected value of the function tolerance.
```

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.

```
problem = load('fleq1');    % Get V, Aeq, beq
V = problem.V; Aeq = problem.Aeq; beq = problem.beq;
norm(Aeq*x-beq,inf)
```

```
ans =
   2.3093e-14
```

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.

Symbolic Math Toolbox Calculates Gradients and Hessians

If you have a Symbolic Math Toolbox license, you can easily calculate analytic gradients and Hessians for objective and constraint functions. There are two relevant Symbolic Math Toolbox functions:

- `jacobian` generates the gradient of a scalar function, and generates a matrix of the partial derivatives of a vector function. So, for example, you can obtain the Hessian matrix, the second derivatives of the objective function, by applying `jacobian` to the gradient. Part of this example shows how to use `jacobian` to generate symbolic gradients and Hessians of objective and constraint functions.

- `matlabFunction` generates either an anonymous function or a file that calculates the values of a symbolic expression. This example shows how to use `matlabFunction` to generate files that evaluate the objective and constraint function and their derivatives at arbitrary points.

Consider the electrostatics problem of placing 10 electrons in a conducting body. The electrons will arrange themselves so as to minimize their total potential energy, subject to the constraint of lying inside the body. It is well known that all the electrons will be on the boundary of the body at a minimum. The electrons are indistinguishable, so there is no unique minimum for this problem (permuting the electrons in one solution gives another valid solution). This example was inspired by Dolan, Moré, and Munson [58].

This example is a conducting body defined by the following inequalities:

$$z \le -|x| - |y|$$

$$x^2 + y^2 + (z+1)^2 \le 1.$$

This body looks like a pyramid on a sphere.
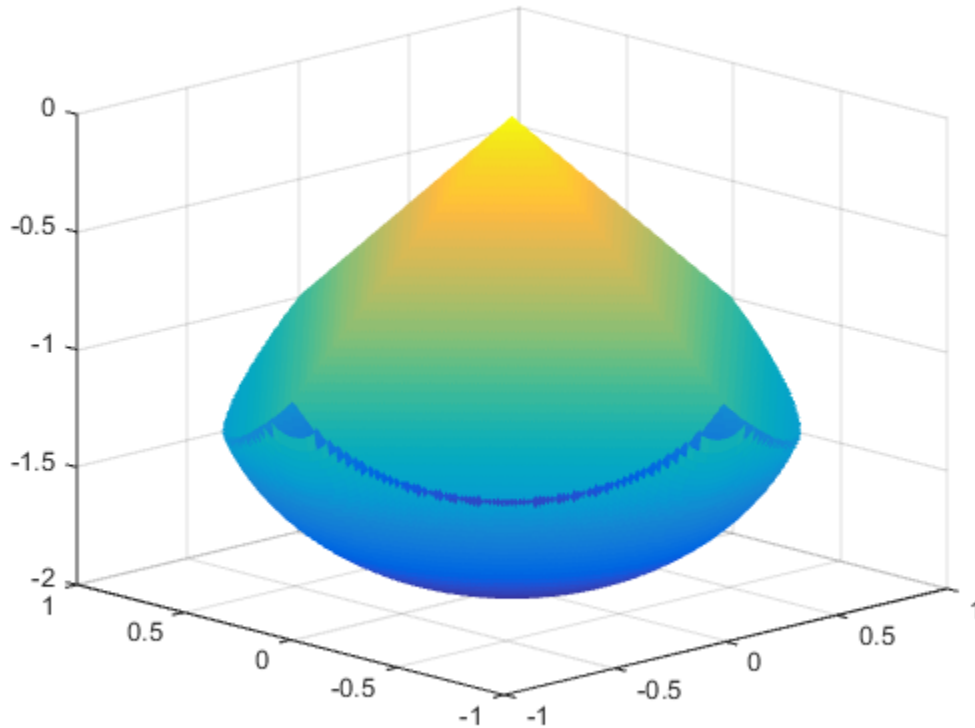
There is a slight gap between the upper and lower surfaces of the figure. This is an artifact of the general plotting routine used to create the figure. This routine erases any rectangular patch on one surface that touches the other surface.

The syntax and structures of the two sets of toolbox functions differ. In particular, symbolic variables are real or complex scalars, but Optimization Toolbox functions pass vector arguments. So there are several steps to take to generate symbolically the objective function, constraints, and all their requisite derivatives, in a form suitable for the interior-point algorithm of `fmincon`:

1 "Create the Variables" on page 6-82
2 "Include the Linear Constraints" on page 6-83
3 "Create the Nonlinear Constraints, Their Gradients and Hessians" on page 6-85

To see the efficiency in using gradients and Hessians, see "Compare to Optimization Without Gradients and Hessians" on page 6-91.

Create the Variables

Generate a symbolic vector `x` as a 30-by-1 vector composed of real symbolic variables `xij`, `i` between 1 and 10, and `j` between 1 and 3. These variables represent the three coordinates of electron `i`: `xi1` corresponds to the x coordinate, `xi2` corresponds to the y coordinate, and `xi3` corresponds to the z coordinate.

```
x = cell(3, 10);
for i = 1:10
    for j = 1:3
        x{j,i} = sprintf('x%d%d',i,j);
    end
end
x = x(:); % now x is a 30-by-1 vector
x = sym(x, 'real');
```

The vector `x` is:

```
x
x =
  x11
  x12
  x13
  x21
  x22
  x23
  x31
  x32
  x33
  x41
```

```
  x42
  x43
  x51
  x52
  x53
  x61
  x62
  x63
  x71
  x72
  x73
  x81
  x82
  x83
  x91
  x92
  x93
 x101
 x102
 x103
```

Include the Linear Constraints

Write the linear constraints in Equation 6-60,
$$z \leq -|x| - |y|,$$

as a set of four linear inequalities for each electron:
$$xi3 - xi1 - xi2 \leq 0$$
$$xi3 - xi1 + xi2 \leq 0$$
$$xi3 + xi1 - xi2 \leq 0$$
$$xi3 + xi1 + xi2 \leq 0$$

Therefore there are a total of 40 linear inequalities for this problem.

Write the inequalities in a structured way:

```
B = [1,1,1;-1,1,1;1,-1,1;-1,-1,1];

A = zeros(40,30);
for i=1:10
    A(4*i-3:4*i,3*i-2:3*i) = B;
end
```

```
b = zeros(40,1);
```

You can see that A*x ≤ b represents the inequalities:

```
A*x
```

```
ans =
    x11 + x12 + x13
    x12 - x11 + x13
    x11 - x12 + x13
    x13 - x12 - x11
    x21 + x22 + x23
    x22 - x21 + x23
    x21 - x22 + x23
    x23 - x22 - x21
    x31 + x32 + x33
    x32 - x31 + x33
    x31 - x32 + x33
    x33 - x32 - x31
    x41 + x42 + x43
    x42 - x41 + x43
    x41 - x42 + x43
    x43 - x42 - x41
    x51 + x52 + x53
    x52 - x51 + x53
    x51 - x52 + x53
    x53 - x52 - x51
    x61 + x62 + x63
    x62 - x61 + x63
    x61 - x62 + x63
    x63 - x62 - x61
    x71 + x72 + x73
    x72 - x71 + x73
    x71 - x72 + x73
    x73 - x72 - x71
    x81 + x82 + x83
    x82 - x81 + x83
    x81 - x82 + x83
    x83 - x82 - x81
    x91 + x92 + x93
    x92 - x91 + x93
    x91 - x92 + x93
    x93 - x92 - x91
 x101 + x102 + x103
 x102 - x101 + x103
```

```
x101 - x102 + x103
x103 - x102 - x101
```

Create the Nonlinear Constraints, Their Gradients and Hessians

The nonlinear constraints in Equation 6-61 ,

$$x^2 + y^2 + (z+1)^2 \le 1,$$

are also structured. Generate the constraints, their gradients, and Hessians as follows:

```
c = sym(zeros(1,10));
i = 1:10;
c = (x(3*i-2).^2 + x(3*i-1).^2 + (x(3*i)+1).^2 - 1).';

gradc = jacobian(c,x).'; % .' performs transpose

hessc = cell(1, 10);
for i = 1:10
    hessc{i} = jacobian(gradc(:,i),x);
end
```

The constraint vector `c` is a row vector, and the gradient of `c(i)` is represented in the ith column of the matrix `gradc`. This is the correct form, as described in "Nonlinear Constraints" on page 2-35.

The Hessian matrices, `hessc{1}`...`hessc{10}`, are square and symmetric. It is better to store them in a cell array, as is done here, than in separate variables such as `hessc1, ..., hesssc10`.

Use the `.'` syntax to transpose. The `'` syntax means conjugate transpose, which has different symbolic derivatives.

Create the Objective Function, Its Gradient and Hessian

The objective function, potential energy, is the sum of the inverses of the distances between each electron pair:

$$\text{energy} = \sum_{i<j} \frac{1}{\left| x_i - x_j \right|}.$$

The distance is the square root of the sum of the squares of the differences in the components of the vectors.

Calculate the energy, its gradient, and its Hessian as follows:

```
energy = sym(0);
for i = 1:3:25
    for j = i+3:3:28
        dist = x(i:i+2) - x(j:j+2);
        energy = energy + 1/sqrt(dist.'*dist);
    end
end

gradenergy = jacobian(energy,x).';

hessenergy = jacobian(gradenergy,x);
```

Create the Objective Function File

The objective function should have two outputs, `energy` and `gradenergy`. Put both functions in one vector when calling `matlabFunction` to reduce the number of subexpressions that `matlabFunction` generates, and to return the gradient only when the calling function (`fmincon` in this case) requests both outputs. This example shows placing the resulting files in your current folder. Of course, you can place them anywhere you like, as long as the folder is on the MATLAB path.

```
currdir = [pwd filesep]; % You may need to use currdir = pwd
filename = [currdir,'demoenergy.m'];
matlabFunction(energy,gradenergy,'file',filename,'vars',{x});
```

This syntax causes `matlabFunction` to return `energy` as the first output, and `gradenergy` as the second. It also takes a single input vector `{x}` instead of a list of inputs `x11, ..., x103`.

The resulting file `demoenergy.m` contains, in part, the following lines or similar ones:

```
function [energy,gradenergy] = demoenergy(in1)
%DEMOENERGY
%   [ENERGY,GRADENERGY] = DEMOENERGY(IN1)
...
x101 = in1(28,:);
...
energy = 1./t140.^(1./2) + ...;
```

```
if nargout > 1
    ...
    gradenergy = [(t174.*(t185 - 2.*x11))./2 - ...];
end
```

This function has the correct form for an objective function with a gradient; see "Writing Scalar Objective Functions" on page 2-18.

Create the Constraint Function File

Generate the nonlinear constraint function, and put it in the correct format.

```
filename = [currdir,'democonstr.m'];
matlabFunction(c,[],gradc,[],'file',filename,'vars',{x},...
    'outputs',{'c','ceq','gradc','gradceq'});
```

The resulting file democonstr.m contains, in part, the following lines or similar ones:

```
function [c,ceq,gradc,gradceq] = democonstr(in1)
%DEMOCONSTR
%    [C,CEQ,GRADC,GRADCEQ] = DEMOCONSTR(IN1)
...
x101 = in1(28,:);
...
c = [t417.^2 + ...];
if nargout > 1
    ceq = [];
end
if nargout > 2
    gradc = [2.*x11,...];
end
if nargout > 3
    gradceq = [];
end
```

This function has the correct form for a constraint function with a gradient; see "Nonlinear Constraints" on page 2-35.

Generate the Hessian Files

To generate the Hessian of the Lagrangian for the problem, first generate files for the energy Hessian and for the constraint Hessians.

The Hessian of the objective function, `hessenergy`, is a very large symbolic expression, containing over 150,000 symbols, as evaluating `size(char(hessenergy))` shows. So it takes a substantial amount of time to run `matlabFunction(hessenergy)`.

To generate a file `hessenergy.m`, run the following two lines:

```
filename = [currdir,'hessenergy.m'];
matlabFunction(hessenergy,'file',filename,'vars',{x});
```

In contrast, the Hessians of the constraint functions are small, and fast to compute:

```
for i = 1:10
    ii = num2str(i);
    thename = ['hessc',ii];
    filename = [currdir,thename,'.m'];
    matlabFunction(hessc{i},'file',filename,'vars',{x});
end
```

After generating all the files for the objective and constraints, put them together with the appropriate Lagrange multipliers in a file `hessfinal.m` as follows:

```
function H = hessfinal(X,lambda)
%
% Call the function hessenergy to start
H = hessenergy(X);

% Add the Lagrange multipliers * the constraint Hessians
H = H + hessc1(X) * lambda.ineqnonlin(1);
H = H + hessc2(X) * lambda.ineqnonlin(2);
H = H + hessc3(X) * lambda.ineqnonlin(3);
H = H + hessc4(X) * lambda.ineqnonlin(4);
H = H + hessc5(X) * lambda.ineqnonlin(5);
H = H + hessc6(X) * lambda.ineqnonlin(6);
H = H + hessc7(X) * lambda.ineqnonlin(7);
H = H + hessc8(X) * lambda.ineqnonlin(8);
H = H + hessc9(X) * lambda.ineqnonlin(9);
H = H + hessc10(X) * lambda.ineqnonlin(10);
```

Run the Optimization

Start the optimization with the electrons distributed randomly on a sphere of radius 1/2 centered at [0,0,–1]:

```
rng default % for reproducibility
Xinitial = randn(3,10); % columns are normal 3-D vectors
for j=1:10
    Xinitial(:,j) = Xinitial(:,j)/norm(Xinitial(:,j))/2;
    % this normalizes to a 1/2-sphere
```

```
end
Xinitial(3,:) = Xinitial(3,:) - 1; % center at [0,0,-1]
Xinitial = Xinitial(:); % Convert to a column vector
```

Set the options to use the interior-point algorithm, and to use gradients and the Hessian:

```
options = optimoptions(@fmincon,'Algorithm','interior-point','GradObj','on',...
    'GradConstr','on','Hessian','user-supplied',...
    'HessFcn',@hessfinal,'Display','final');
```

Call fmincon:

```
[xfinal fval exitflag output] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options)
```

The output is as follows:

```
Local minimum found that satisfies the constraints.

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

xfinal =
   -0.0317
    0.0317
   -1.9990
    0.6356
   -0.6356
   -1.4381
    0.0000
   -0.0000
   -0.0000
    0.0000
   -1.0000
   -1.0000
    1.0000
   -0.0000
   -1.0000
   -1.0000
   -0.0000
   -1.0000
    0.6689
    0.6644
   -1.3333
   -0.6667
    0.6667
   -1.3333
    0.0000
    1.0000
   -1.0000
   -0.6644
   -0.6689
   -1.3333

fval =
```

```
        34.1365

exitflag =
        1

output =
          iterations: 19
           funcCount: 28
      constrviolation: 0
            stepsize: 4.0372e-05
           algorithm: 'interior-point'
       firstorderopt: 4.0015e-07
        cgiterations: 55
             message: 'Local minimum found that satisfies the constraints.…'
```

Even though the initial positions of the electrons were random, the final positions are nearly symmetric:
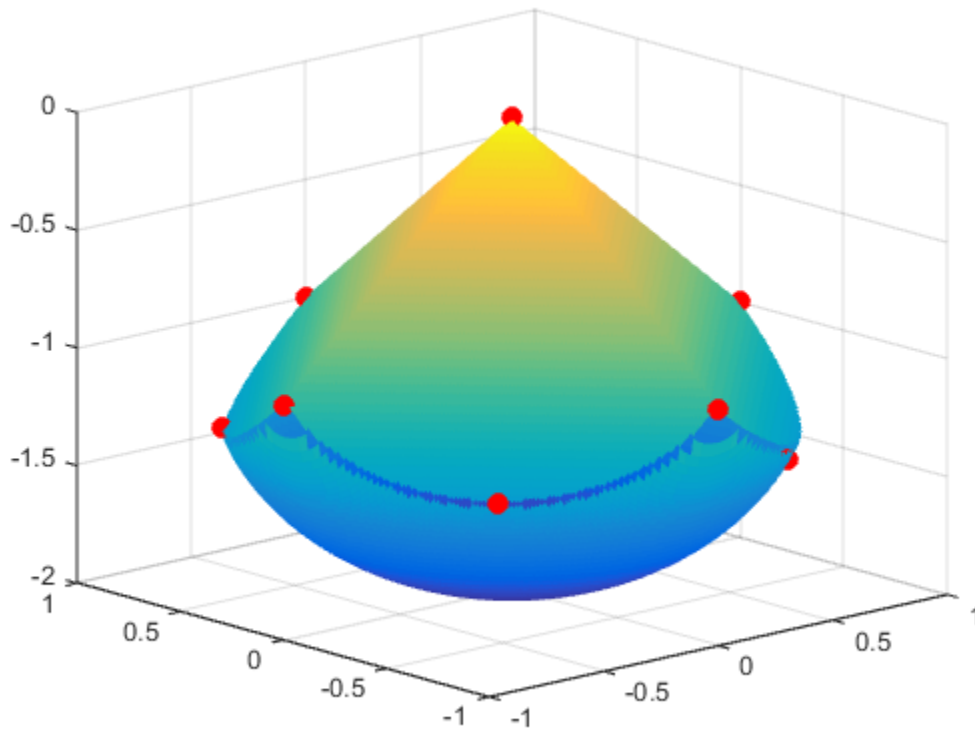
Compare to Optimization Without Gradients and Hessians

The use of gradients and Hessians makes the optimization run faster and more accurately. To compare with the same optimization using no gradient or Hessian information, set the options not to use gradients and Hessians:

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
    'Display','final');
[xfinal2 fval2 exitflag2 output2] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options)
```

The output shows that `fmincon` found an equivalent minimum, but took more iterations and many more function evaluations to do so:

```
Local minimum found that satisfies the constraints.

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

xfinal2 =
    0.0000
    1.0000
   -1.0000
    0.6689
   -0.6644
   -1.3334
   -0.6644
    0.6689
   -1.3334
    0.0000
   -1.0000
   -1.0000
    0.6357
    0.6357
   -1.4380
   -0.0317
   -0.0317
   -1.9990
    1.0000
    0.0000
   -1.0000
   -1.0000
    0.0000
   -1.0000
    0.0000
    0.0000
   -0.0000
   -0.6667
   -0.6667
   -1.3334

fval2 =
   34.1365

exitflag2 =
    1
```

```
output2 =
        iterations: 77
         funcCount: 2431
    constrviolation: 0
          stepsize: 6.0588e-07
         algorithm: 'interior-point'
     firstorderopt: 2.9894e-06
       cgiterations: 0
           message: 'Local minimum found that satisfies the constraints.…'
```

In this run the number of function evaluations (in `output2.funcCount`) is 2431, compared to 28 (in `output.funcCount`) when using gradients and Hessian.

Clear the Symbolic Variable Assumptions

The symbolic variables in this example have the assumption, in the symbolic engine workspace, that they are real. To clear this assumption from the symbolic engine workspace, it is not sufficient to delete the variables. You must clear the variables using the syntax

```
syms x11 x12 x13 clear
```

or reset the symbolic engine using the command

```
reset(symengine)
```

After resetting the symbolic engine you should clear all symbolic variables from the MATLAB workspace with the `clear` command, or `clear` *variable_list*.

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 a file that computes the objective function.

```
function f = myfun(x,s)
% Objective function
f = sum((x-0.5).^2);
```

Second, write a 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
x =
    0.6675
    0.3012
    0.4022
```

The function value and the maximum values of the semi-infinite constraints at the solution x are

```
fval
fval =
    0.0771

[c,ceq,K1,K2] = mycon(x,NaN); % Initial sampling interval
max(K1)
ans =
    -0.0077
max(K2)
ans =
    -0.0812
```

A plot of the semi-infinite constraints is produced.
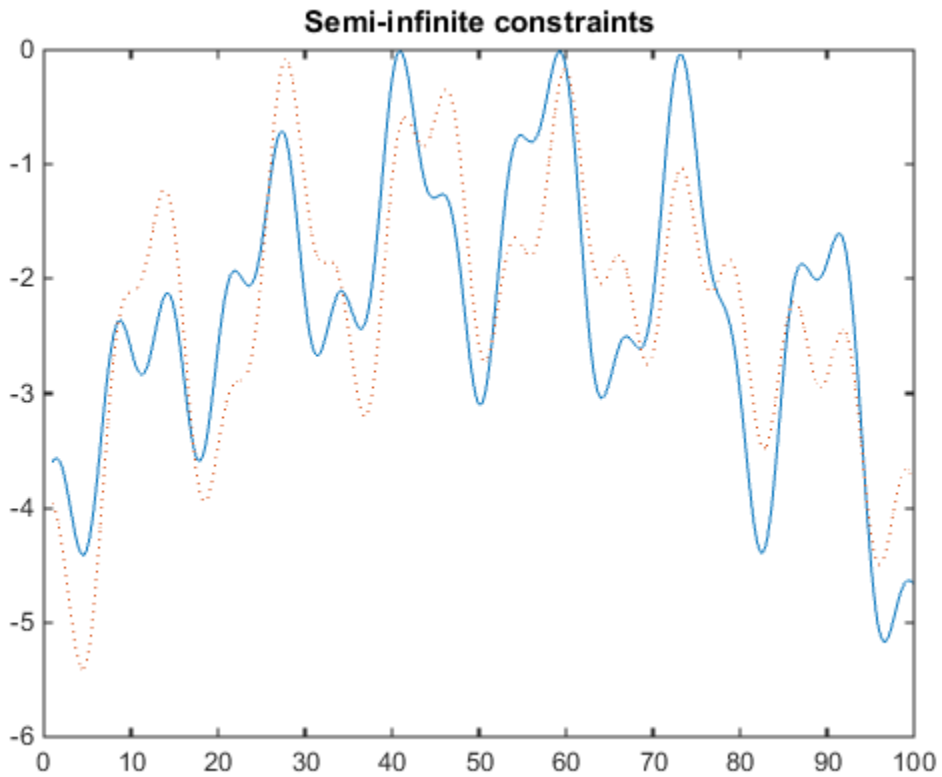
This plot shows how peaks in both constraints are on the constraint boundary.

The plot command inside `mycon.m` slows down the computation. Remove this line to improve the speed.

Two-Dimensional 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

$$K_1(x,w) = \sin(w_1 x_1)\cos(w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \ldots$$
$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1.5,$$

for all values of w_1 and w_2 over the ranges
$1 \le w_1 \le 100,$
$1 \le w_2 \le 100,$

starting at the point $x = [0.25, 0.25, 0.25]$.

Note that the semi-infinite constraint is two-dimensional, that is, a matrix.

First, write a file that computes the objective function.

```
function f = myfun(x,s)
% Objective function
f = sum((x-0.2).^2);
```

Second, write a 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
x =
    0.2522    0.1714    0.1936
```

and the function value at the solution is

```
fval
fval =
    0.0036
```

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 =
    -0.0332
```

This call to mycon produces the following surf plot, which shows the semi-infinite constraint at x.

Semi-infinite constraint

7
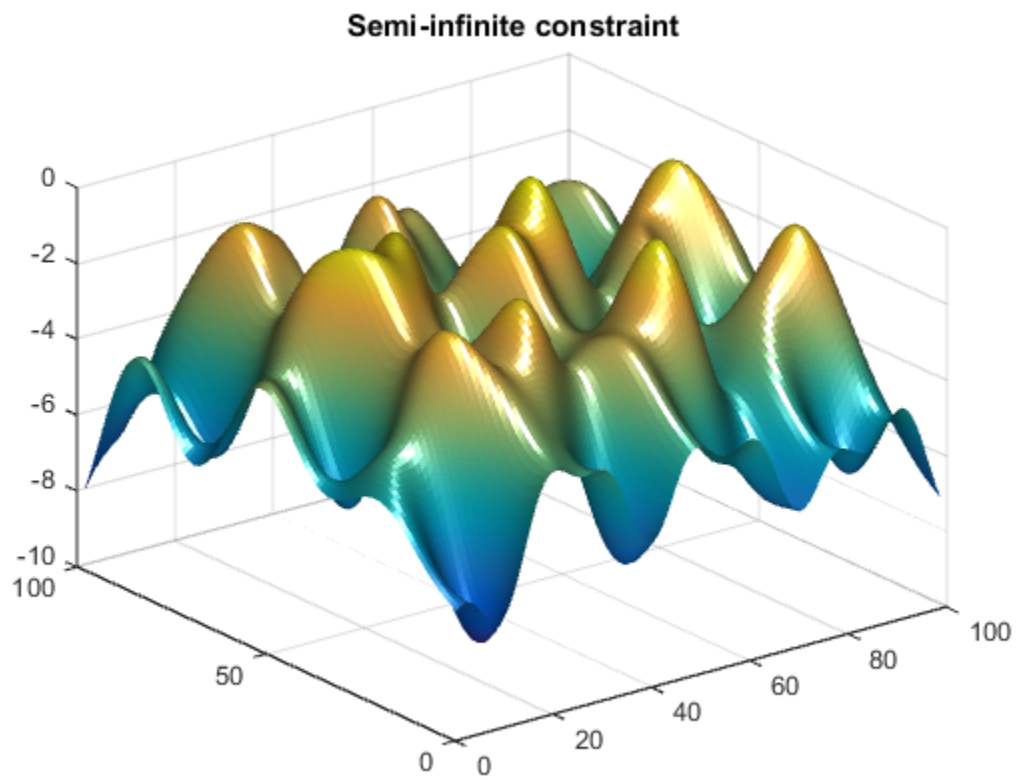
Multiobjective Algorithms and Examples

Multiobjective Optimization Algorithms

In this section...

Multiobjective Optimization 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

  $$\frac{F_i(x) - F_i^*}{w_i}.$$

  This minimization is supposed to be accomplished while satisfying all types of constraints: $c(x) \leq 0$, $ceq(x) = 0$, $A\,x \leq b$, $Aeq\,x = beq$, and $l \leq x \leq u$.

  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 7-1. This implies the expression that is the formal statement of the goal attainment problem:

$$\min_{x,\gamma} \gamma$$

such that $F(x) - w\,\gamma \le F^*$, $c(x) \le 0$, $ceq(x) = 0$, $A\,x \le b$, $Aeq\,x = beq$, 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 $c(x) \le 0$, $ceq(x) = 0$, $A\,x \le b$, $Aeq\,x = beq$, 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, $F^* = \left\{ F^*_1, F^*_2, ..., F^*_m \right\}$, associated with a set of objectives, $F(x) = \{F_1(x), F_2(x), ..., F_m(x)\}$. 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, $w = \{w_1, w_2, ..., w_m\}$, and is expressed as a standard optimization problem using the formulation

$$\underset{\gamma \in \Re,\ x \in \Omega}{\text{minimize}} \gamma$$

such that $F_i(x) - w_i \gamma \le F^*_i$, $\ i = 1, ..., m$.

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.

$$\underset{\gamma, x \in \Omega}{\textbf{minimize}} \ \gamma \quad \text{subject to} \quad F_1(x) - w_1\gamma \le F_1^*$$
$$F_2(x) - w_2\gamma \le F_2^*$$
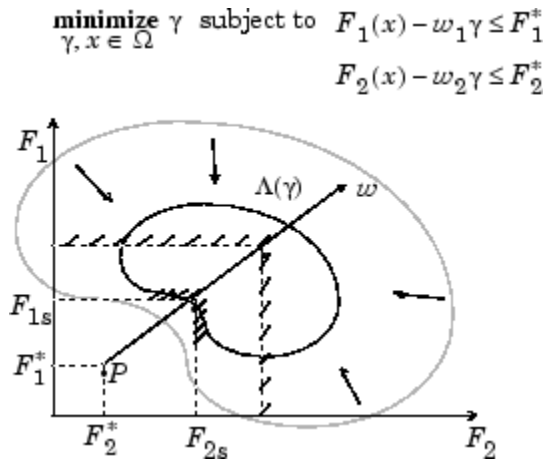
Figure 7-1. Geometrical Representation of the Goal Attainment Method

Specification of the goals, $\left\{ F_1^*, F_2^* \right\}$, 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 7-2 as the minimax problem

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

where

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

Following the argument of Brayton et al. [2] for minimax optimization using SQP, using the merit function of Equation 6-46 for the goal attainment problem of Equation 7-3 gives

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

When the merit function of Equation 7-4 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 $\max \Lambda_i$ 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 $\max \Lambda_i$ 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.,

$$\psi(x) = \max_i \Lambda_i.$$

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 7-5 then becomes infinite for arbitrary violations of the constraints.

To overcome this problem while still retaining the features of Equation 7-5, the merit function is combined with that of Equation 6-47, giving the following:

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

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 6-31.

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 6-46.

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.

Using `fminimax` with a Simulink Model

Another approach to optimizing the control parameters in the Simulink model shown in Plant with Actuator Saturation is to use the `fminimax` function. In this case, rather than minimizing the error between the output and the input signal, you minimize the maximum value of the output at any time `t` between 0 and 100.

The code for this example, shown below, is contained in the function `runtrackmm`, in which the objective function is simply the output `yout` returned by the `sim` command. But minimizing the maximum output at all time steps might force the output to be far below unity for some time steps. To keep the output above `0.95` after the first `20` seconds, the constraint function `trackmmcon` contains the constraint `yout >= 0.95` from `t=20` to `t=100`. Because constraints must be in the form `g ≤ 0`, the constraint in the function is `g = -yout(20:100)+.95`.

Both `trackmmobj` and `trackmmcon` use the result `yout` from `sim`, calculated from the current PID values. To avoid calling the simulation twice, `runtrackmm` has nested functions so that the value of `yout` is shared between the objective and constraint functions. The simulation is called only when the current point changes.

The following is the code for `runtrackmm`:

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

optsim % initialize Simulink(R)
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
pold = []; % tracks last pid
options = optimoptions('fminimax','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.
        updateIfNeeded(pid)
```

```
        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
        updateIfNeeded(pid)

        c = -yout(20:100)+.95;
        ceq=[];
    end

    function updateIfNeeded(pid)
        if ~isequal(pid,pold) % compute only if needed

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

            myobj = sim('optsim','SrcWorkspace','Current');
            yout = myobj.get('yout');

            pold = pid;
        end
    end

end
```

Copy the code for `runtrackmm` to a file named `runtrackmm.m`, placed in a folder on your MATLAB path.

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

```
[Kp,Ki,Kd] = runtrackmm
Done initializing optsim.

             Objective       Max     Line search    Directional
Iter F-count     value    constraint  steplength     derivative   Procedure
   0     5           0      1.11982
   1    11       1.184      0.07978           1           0.482
   2    17       1.012      0.04285           1          -0.236
   3    23      0.9995     0.007058           1         -0.0186    Hessian modified twice
   4    29      0.9997    9.705e-07           1         0.00716    Hessian modified

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
```

```
twice the selected value of the step size tolerance and constraints are
satisfied to within the default value of the constraint tolerance.

Kp =
    0.5910

Ki =
    0.0606

Kd =
    5.5383
```

The last value in the `Objective value` column of the output shows that the maximum value for all the time steps is `0.9997`. The closed loop response with this result is shown in the figure Closed-Loop Response Using fminimax.

This solution differs from the solution obtained in "lsqnonlin with a Simulink Model" on page 10-11 because you are solving different problem formulations.
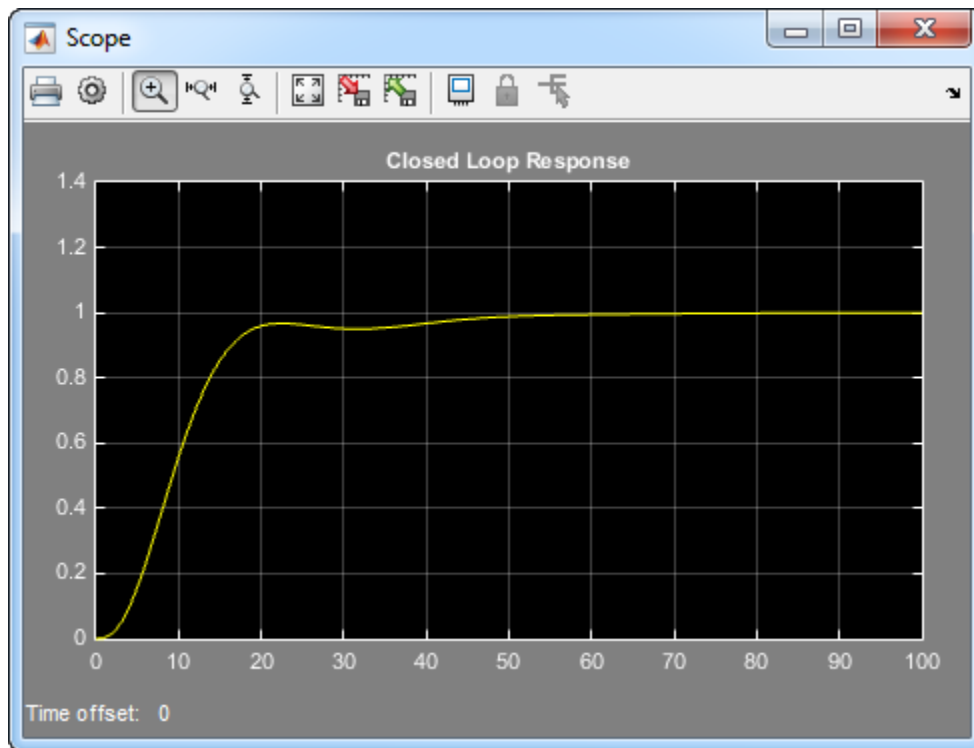
Closed-Loop Response Using fminimax

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

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

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, `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 uses `a`, the magnitude function coefficients, and `w`, the discretization of the frequency domain of interest.

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 7-3). 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 = optimoptions('fgoalattain','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 a 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,'-.b');
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 = optimoptions('fgoalattain','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). Note

that you could use the `firpm` function in Signal Processing Toolbox™ software to design this filter.
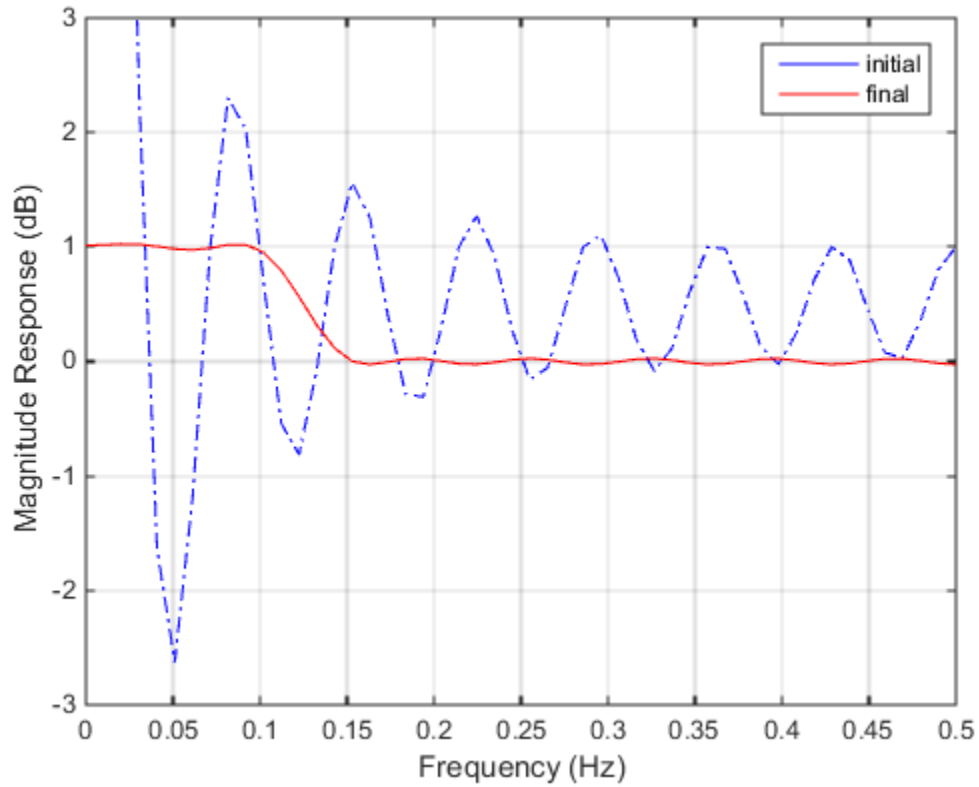
Magnitude Response with Initial and Final Magnitude Coefficients

Generate and Plot a Pareto Front

This example shows how to generate and plot a Pareto front for a 2-D multiobjective function using `fgoalattain`.

The two objectives in this example are shifted and scaled versions of the convex function $\sqrt{1+x^2}$.

```
function f = simple_mult(x)

f(:,1) = sqrt(1+x.^2);
f(:,2) = 4 + 2*sqrt(1+(x-1).^2);
```

Both components are increasing as x decreases below 0 or increases above 1. In between 0 and 1, $f_1(x)$ is increasing and $f_2(x)$ is decreasing, so there is a tradeoff region.

```
t = linspace(-0.5,1.5);
F = simple_mult(t);
plot(t,F,'LineWidth',2)
hold on
plot([0,0],[0,8],'g--');
plot([1,1],[0,8],'g--');
plot([0,1],[1,6],'k.','MarkerSize',15);
text(-0.25,1.5,'Minimum(f_1(x))')
text(.75,5.5,'Minimum(f_2(x))')
hold off
legend('f_1(x)','f_2(x)')
xlabel({'x';'Tradeoff region between the green lines'})
```

Tradeoff region between the green lines
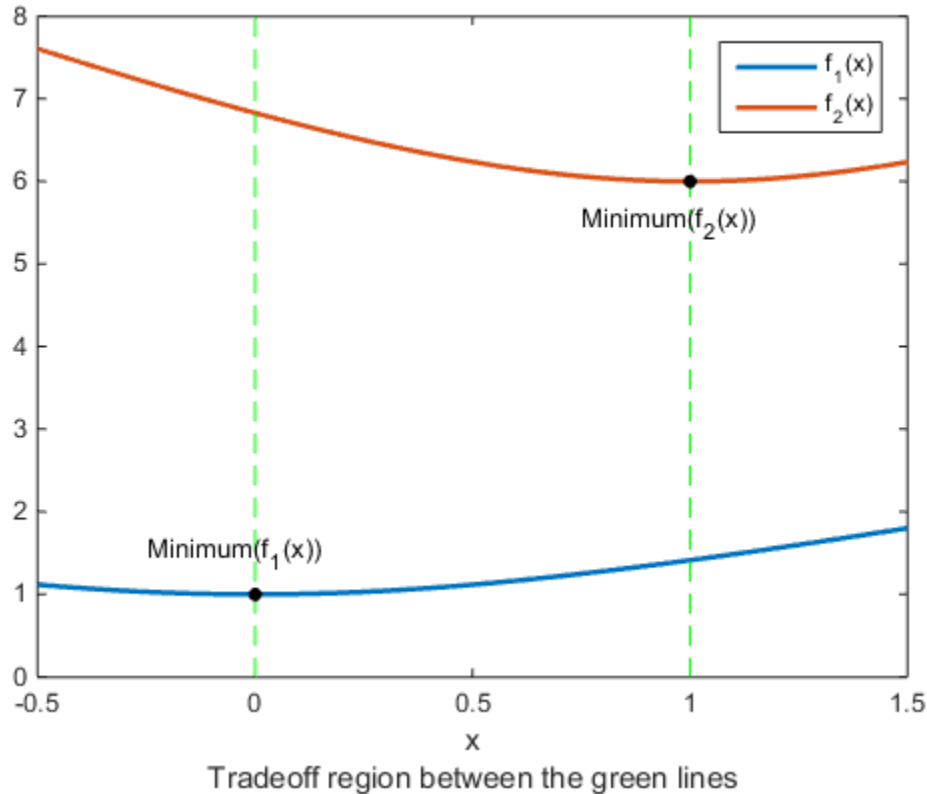
To find the Pareto front, first find the unconstrained minima of the two functions. In this case, you can see by inspection that the minimum of $f_1(x)$ is 1, and the minimum of $f_2(x)$ is 6, but in general you might need to use an optimization routine.

In general, write a function that returns a particular component of the multiobjective function.

```
function z = pickindex(x,k)
z = simple_mult(x); % evaluate both objectives
z = z(k); % return objective k
```

Then find the minimum of each component using an optimization solver. You can use `fminbnd` in this case, or `fminunc` for higher-dimensional problems.

```
k = 1;
[min1,minfn1] = fminbnd(@(x)pickindex(x,k),-1,2);
k = 2;
[min2,minfn2] = fminbnd(@(x)pickindex(x,k),-1,2);
```

Set goals that are the unconstrained optima for each component. You can simultaneously achieve these goals only if the multiobjective functions do not interfere with each other, meaning there is no tradeoff.

```
goal = [minfn1,minfn2];
```

To calculate the Pareto front, take weight vectors [a,1–a] for a from 0 through 1. Solve the goal attainment problem, setting the weights to the various values.

```
nf = 2; % number of objective functions
N = 50; % number of points for plotting
onen = 1/N;
x = zeros(N+1,1);
f = zeros(N+1,nf);
fun = @simple_mult;
x0 = 0.5;
options = optimoptions('fgoalattain','Display','off');
for r = 0:N
    t = onen*r; % 0 through 1
    weight = [t,1-t];
    [x(r+1,:),f(r+1,:)] = fgoalattain(fun,x0,goal,weight,...
        [],[],[],[],[],[],[],options);
end

figure
plot(f(:,1),f(:,2),'k.');
xlabel('f_1')
ylabel('f_2')
```

You can see the tradeoff between the two objectives.

8

Linear Programming and Mixed-Integer Linear Programming

Linear Programming Algorithms

Linear Programming 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:
$A\, x \le b$
$Aeq\, x = beq$
$l \le x \le u.$

Interior-Point `linprog` Algorithm

The `linprog` `'interior-point'` algorithm is very similar to the "`interior-point-convex quadprog` Algorithm" on page 9-2. It also shares many features with the `linprog` `'interior-point-legacy'` algorithm. These algorithms have the same general outline:

1 Presolve, meaning simplification and conversion of the problem to a standard form.

2 Generate an initial point. The choice of an initial point is especially important for solving interior-point algorithms efficiently, and this step can be time-consuming.

3 Predictor-corrector iterations to solve the KKT equations. This step is generally the most time-consuming.

Presolve

The algorithm begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

· Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.

· Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.

· Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.

· Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.

· Check if the bounds and linear constraints are consistent.

· Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.

· Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For simplicity, if the problem is not solved in the presolve step, the algorithm shifts all finite lower bounds to zero.

Generate Initial Point

To set the initial point, x0, the algorithm does the following.

1 Initialize x0 to ones(n,1), where n is the number of elements of the objective function vector *f*.

2 Convert all bounded components to have a lower bound of 0. If component `i` has a finite upper bound `u(i)`, then `x0(i) = u/2`.

3 For components that have only one bound, modify the component if necessary to lie strictly inside the bound.

4 To put `x0` close to the central path, take one predictor-corrector step, and then modify the resulting position and slack variables to lie well within any bounds. For details of the central path, see Nocedal and Wright [6], page 397.

Predictor-Corrector

Similar to the `fmincon` interior-point algorithm, the `interior-point-convex` algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) conditions hold. To describe these equations for the linear programming problem, consider the standard form of the linear programming problem after preprocessing:

$$\min_x f^T x \text{ subject to } \begin{cases} \bar{A}x = \bar{b} \\ x + t = u \\ x, t \geq 0. \end{cases}$$

- Assume for now that all variables have at least one finite bound. By shifting and negating components, if necessary, this assumption means that all x components have a lower bound of 0.

- $\bar{A}$ is the extended linear matrix that includes both linear inequalities and linear equalities. $\bar{b}$ is the corresponding linear equality vector. $\bar{A}$ also includes terms for extending the vector x with slack variables s that turn inequality constraints to equality constraints:

$$\bar{A}x = \begin{pmatrix} A_{eq} & 0 \\ A & I \end{pmatrix} \begin{pmatrix} x_0 \\ s \end{pmatrix},$$

where x_0 means the original x vector.

- t is the vector of slacks that convert upper bounds to equalities.

The Lagrangian for this system involves the following vectors:

- y, Lagrange multipliers associated with the linear equalities

- v, Lagrange multipliers associated with the lower bound (positivity constraint)
- w, Lagrange multipliers associated with the upper bound

The Lagrangian is

$$L = f^T x - y^T (\bar{A} x - \bar{b}) - v^T x - w^T (u - x - t).$$

Therefore, the KKT conditions for this system are

$$f - \bar{A}^T y - v + w = 0$$
$$\bar{A} x = \bar{b}$$
$$x + t = u$$
$$v_i x_i = 0$$
$$w_i t_i = 0$$
$$(x, v, w, t) \geq 0.$$

The algorithm first predicts a step from the Newton-Raphson formula, and then computes a corrector step. The corrector attempts to reduce the residual in the nonlinear complementarity equations $s_i z_i = 0$. The Newton-Raphson step is

$$
\begin{pmatrix}
0 & -\bar{A}^T & 0 & -I & I \\
\bar{A} & 0 & 0 & 0 & 0 \\
-I & 0 & -I & 0 & 0 \\
V & 0 & 0 & X & 0 \\
0 & 0 & W & 0 & T
\end{pmatrix}
\begin{pmatrix}
\Delta x \\
\Delta y \\
\Delta t \\
\Delta v \\
\Delta w
\end{pmatrix}
= -
\begin{pmatrix}
f - \bar{A}^T y - v + w \\
\bar{A} x - \bar{b} \\
u - x - t \\
VX \\
WT
\end{pmatrix}
= -
\begin{pmatrix}
r_d \\
r_p \\
r_{ub} \\
r_{vx} \\
r_{wt}
\end{pmatrix},
$$

where X, V, W, and T are diagonal matrices corresponding to the vectors x, v, w, and t respectively. The residual vectors on the far right side of the equation are:

- r_d, the dual residual
- r_p, the primal residual
- r_{ub}, the upper bound residual
- r_{vx}, the lower bound complementarity residual
- r_{wt}, the upper bound complementarity residual

To solve Equation 8-1, first convert it to the symmetric matrix form

$$
\begin{pmatrix} -D & \bar{A}^T \\ \bar{A} & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} -R \\ r_p \end{pmatrix},
$$

where

$$
D = X^{-1}V + T^{-1}W
$$
$$
R = -r_d - X^{-1}r_{vx} + T^{-1}r_{wt} + T^{-1}Wr_{ub}.
$$

All the matrix inverses in the definitions of D and R are simple to compute because the matrices are diagonal.

To derive Equation 8-2 from Equation 8-1, notice that the second row of Equation 8-2 is the same as the second matrix row of Equation 8-1. The first row of Equation 8-2 comes from solving the last two rows of Equation 8-1 for Δv and Δw, and then solving for Δt.

Equation 8-2 is symmetric, but it is not positive definite because of the $-D$ term. Therefore, you cannot solve it using a Cholesky factorization. A few more steps lead to a different equation that is positive definite, and hence can be solved efficiently by Cholesky factorization.

The second set of rows of Equation 8-2 is

$$
\bar{A}\Delta y = r_p
$$

and the first set of rows is

$$
-D\Delta x + \bar{A}^T \Delta y = -R.
$$

Substituting

$$
\Delta x = D^{-1}A^T \Delta y + D^{-1}R
$$

gives

$$\bar{A}D^{-1}\bar{A}^T\Delta y = \bar{A}D^{-1}R + r_p.$$

Usually, the most efficient way to find the Newton step is to solve Equation 8-3 for Δy using Cholesky factorization. Cholesky factorization is possible because the matrix multiplying Δy is obviously symmetric and, in the absence of degeneracies, is positive definite. Afterwards, to find the Newton step, back substitute to find Δx, Δt, Δv, and Δw.

However, when $\bar{A}$ has a dense column, it can be more efficient to solve Equation 8-2 instead. The `linprog` interior-point algorithm chooses the solution algorithm based on the density of columns.

For more algorithm details, see Mehrotra [5].

After calculating the corrected Newton step, the algorithm performs more calculations to get both a longer current step, and to prepare for better subsequent steps. These multiple correction calculations can improve both performance and robustness. For details, see Gondzio [3].

Stopping Conditions

The predictor-corrector algorithm iterates until it reaches a point that is feasible (satisfies the constraints to within tolerances) and where the relative step sizes are small. Specifically, define

$$\rho = \max\left(1, \|\bar{A}\|, \|f\|, \|\bar{b}\|\right).$$

The algorithm stops when all of these conditions are satisfied:

$$\|r_p\|_\infty \le \rho\,\mathrm{TolCon}$$
$$\|r_d\|_\infty \le \rho\,\mathrm{TolFun}$$
$$r_c \le \mathrm{TolFun},$$

where

$$r_c = \max_i\left(\min\left(|x_i v_i|, |x_i|, |v_i|\right), \min\left(|t_i w_i|, |t_i|, |w_i|\right)\right).$$

r_c essentially measures the size of the complementarity residuals xv and tw, which are each vectors of zeros at a solution.

Interior-Point-Legacy Linear Programming

Introduction

The default interior-point-legacy 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 8-10). After preprocessing, the problem has the form

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \le x \le u. \end{cases}$$

The upper bounds constraints are implicitly included in the constraint matrix A. With the addition of primal slack variables s, Equation 8-4 becomes

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x = b \\ x + s = u \\ x \ge 0, \ s \ge 0. \end{cases}$$

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

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \ge 0, \ w \ge 0, \end{cases}$$

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 8-5 and the dual Equation 8-6, are

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

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

where $x_i z_i$ and $s_i w_i$ denote component-wise multiplication.

The quadratic equations $x_i z_i = 0$ and $s_i w_i = 0$ are called the *complementarity* conditions for the linear program; the other (linear) equations are called the *feasibility* conditions. The quantity

$x^T z + s^T w$

is the *duality gap*, which measures the residual of the complementarity portion of F when $(x,z,s,w) \geq 0$.

The algorithm is a *primal-dual algorithm*, meaning that both the primal and the dual programs are solved simultaneously. It can be considered a Newton-like method, applied to the linear-quadratic system $F(x,y,z,s,w) = 0$ in Equation 8-7, 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 8-5.)

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 predictor/corrector directions, 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 system 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 loops until the iterates converge. The main stopping criteria is a standard one:

$$\frac{\|r_b\|}{\max\left(1,\|b\|\right)} + \frac{\|r_f\|}{\max\left(1,\|f\|\right)} + \frac{\|r_u\|}{\max\left(1,\|u\|\right)} + \frac{\left|f^T x - b^T y + u^T w\right|}{\max\left(1, \left|f^T x\right|, \left|b^T y - u^T w\right|\right)} \le tol,$$

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 8-7.

Preprocessing

The algorithm begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

- Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.

- Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.

- Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.

- Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.

- Check if the bounds and linear constraints are consistent.

- Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.

- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For simplicity, the algorithm shifts all lower bounds to zero.

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 `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 6-28). 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 ith row of the m-by-n matrix A.

$$\min_{d \in \Re^n} q(d) = c^T d,$$
$$A_i d = b_i, \quad i = 1, \ldots, m_e$$
$$A_i d \le b_i, \quad i = m_e + 1, \ldots, m.$$

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

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

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

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

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c.$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c.$$

A step is then taken of the form

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

At each iteration, because of the quadratic nature of the objective function, there are only two choices of step length a. 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 8-9). 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

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

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

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

$$\bar{A}_k^T \lambda_k = c.$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 8-9). 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

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

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 8-15 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).

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

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

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

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.

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, $f^T x$, at each

step. This section describes an improved version of the original simplex algorithm that returns a vertex optimal solution.

This section covers the following topics:

- "Main Algorithm" on page 8-16
- "Preprocessing" on page 8-17
- "Using the Simplex Algorithm" on page 8-18
- "Basic and Nonbasic Variables" on page 8-18

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 8-18 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

$$\min_x \sum_j P_j \;\; \text{subject to} \; \begin{cases} A \cdot x \le b \\ Aeq \cdot x = beq. \end{cases}$$

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 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 8-18 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.

The algorithm detects when there is no progress in the Phase 2 solution process. It attempts to continue by performing bound perturbation. For an explanation of this part of the algorithm, see Applegate, Bixby, Chvatal, and Cook [59].

Preprocessing

The simplex algorithm uses the same preprocessing steps as the interior-point linear programming solver, which are described in "Preprocessing" on page 8-10. 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 $a\,x = b$, where a is a row of *Aeq*, the algorithm computes the lower and upper bounds of the linear combination $a\,x$ 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 $a\,x$ 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 the `Algorithm` option to `'simplex'` using `optimoptions`.

```
options = optimoptions(@linprog,'Algorithm','simplex');
```

Then call `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 $A\,x = b$ 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*.

Dual-Simplex Algorithm

At a high level, the `linprog` `'dual-simplex'` algorithm essentially performs a simplex algorithm on the *dual problem*.

The algorithm begins by preprocessing as described in "Preprocessing" on page 8-10. For details, see Andersen and Andersen [1] and Nocedal and Wright [6], Chapter 13. This preprocessing reduces the original linear programming problem to the form of Equation 8-4:

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \le x \le u. \end{cases}$$

A and b are transformed versions of the original constraint matrices. This is the primal problem.

As explained in Equation 8-6, the dual problem is to find vectors y and w, and a slack variable vector z that solve

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \ge 0, \ w \ge 0. \end{cases}$$

It is well known (for example, see [6]) that any finite solution of the dual problem gives a solution to the primal problem, and any finite solution of the primal problem gives a solution of the dual problem. Furthermore, if either the primal or dual problem is unbounded, then the other problem is infeasible. And if either the primal or dual problem

is infeasible, then the other problem is either infeasible or unbounded. Therefore, the two problems are equivalent in terms of obtaining a finite solution, if one exists. Because the primal and dual problems are mathematically equivalent, but the computational steps differ, it can be better to solve the primal problem by solving the dual problem.

To help alleviate degeneracy (see Nocedal and Wright [6], page 366), the dual simplex algorithm begins by perturbing the objective function.

Phase 1 of the dual simplex algorithm is to find a dual feasible point. The algorithm does this by solving an auxiliary linear programming problem, similar to "Phase 1" on page 8-16 for the simplex algorithm.

During Phase 2, the solver repeatedly chooses an entering variable and a leaving variable, analogously to "Phase 2" on page 8-17 for the primal simplex algorithm. The algorithm chooses a leaving variable according to a technique suggested by Forrest and Goldfarb [2] called dual steepest-edge pricing. The algorithm chooses an entering variable using the variation of Harris' ratio test suggested by Koberstein [4]. To help alleviate degeneracy, the algorithm can introduce additional perturbations during Phase 2.

The solver iterates, attempting to maintain dual feasibility while reducing primal infeasibility, until the solution to the reduced, perturbed problem is both primal feasible and dual feasible. The algorithm unwinds the perturbations that it introduced. If the solution (to the perturbed problem) is dual infeasible for the unperturbed (original) problem, then the solver restores dual feasibility using primal simplex or Phase 1 algorithms. Finally, the solver unwinds the preprocessing steps to return the solution to the original problem.

References

[1] Andersen, E. D., and K. D. Andersen. *Presolving in linear programming*. Math. Programming 71, 1995, pp. 221–245.

[2] Forrest, J. J., and D. Goldfarb. *Steepest-edge simplex algorithms for linear programming*. Math. Programming 57, 1992, pp. 341–374.

[3] Gondzio, J. "Multiple centrality corrections in a primal dual method for linear programming." *Computational Optimization and Applications*, Volume 6, Number 2, 1996, pp. 137–156. Available at http://www.maths.ed.ac.uk/~gondzio/software/correctors.ps.

[4] Koberstein, A. *Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation*. Computational Optim. and Application 41, 2008, pp. 185–204.

[5] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization*, Vol. 2, 1992, pp 575–601.

[6] Nocedal, J., and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer-Verlag, 2006.

Typical Linear Programming Problem

This example shows the solution of a typical linear programming problem. 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.

Use `linprog` to solve the problem:

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

Because the iterative display was set using `optimoptions`, 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+03 2.19e+01 1.91e+04 1.00e+02
  Iter    1:  1.15e+02 3.16e-15 3.62e+03 9.90e-01
  Iter    2:  9.79e-13 2.62e-15 4.32e+02 9.48e-01
  Iter    3:  3.49e-12 5.93e-15 7.78e+01 6.88e-01
  Iter    4:  4.86e-11 8.35e-16 2.38e+01 2.69e-01
  Iter    5:  2.18e-10 3.39e-16 5.05e+00 6.89e-02
  Iter    6:  1.05e-10 9.55e-17 1.64e-01 2.34e-03
  Iter    7:  9.43e-12 1.51e-16 1.09e-05 1.55e-07
  Iter    8:  1.11e-12 1.68e-16 1.09e-11 1.52e-13
Optimization terminated.
```

For this problem, the interior-point 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,fval,output

exitflag =
     1

fval =
  -70.0000

output =
          iterations: 8
           algorithm: 'interior-point-legacy'
        cgiterations: 0
             message: 'Optimization terminated.'
      constrviolation: 4.8317e-13
        firstorderopt: 2.7908e-13
```

Maximize Long-Term Investments Using Linear Programming

This example shows how to use the `linprog` solver in Optimization Toolbox® to solve an investment problem with deterministic returns over a fixed number of years T. The problem is to allocate your money over available investments to maximize your final wealth.

Problem Formulation

Suppose that you have an initial amount of money `Capital_0` to invest over a time period of T years in N zero-coupon bonds. Each bond pays an interest rate that compounds each year, and pays the principal plus compounded interest at the end of a maturity period. The objective is to maximize the total amount of money after T years.

You can include a constraint that no single investment is more than a certain fraction of your total capital.

This example shows the problem setup on a small case first, and then formulates the general case.

You can model this as a linear programming problem. Therefore, to optimize your wealth, formulate the problem for solution by the `linprog` solver.

Introductory Example

Start with a small example:

- The starting amount to invest `Capital_0` is $1000.
- The time period T is 5 years.
- The number of bonds N is 4.
- To model uninvested money, have one option B0 available every year that has a maturity period of 1 year and a interest rate of 0%.
- Bond 1, denoted by B1, can be purchased in year 1, has a maturity period of 4 years, and interest rate of 2%.
- Bond 2, denoted by B2, can be purchased in year 5, has a maturity period of 1 year, and interest rate of 4%.
- Bond 3, denoted by B3, can be purchased in year 2, has a maturity period of 4 years, and interest rate of 6%.

- Bond 4, denoted by B4, can be purchased in year 2, has a maturity period of 3 years, and interest rate of 6%.

By splitting up the first option B0 into 5 bonds with maturity period of 1 year and interest rate of 0%, this problem can be equivalently modeled as having a total of 9 available bonds, such that for k=1..9

- Entry k of vector PurchaseYears represents the year that bond k is available for purchase.
- Entry k of vector Maturity represents the maturity period m_k of bond k.
- Entry k of vector InterestRates represents the interest rate ρ_k of bond k.

Visualize this problem by horizontal bars that represent the available purchase times and durations for each bond.

```
% Time period in years
T = 5;
% Number of bonds
N = 4;
% Initial amount of money
Capital_0 = 1000;
% Total number of buying oportunities
nPtotal = N+T;
% Purchase times
PurchaseYears = [1;2;3;4;5;1;5;2;2];
% Bond durations
Maturity = [1;1;1;1;1;4;1;4;3];
% Interest rates
InterestRates = [0;0;0;0;0;2;4;6;6];

plotInvestments(N,PurchaseYears,Maturity,InterestRates)
```

	Year 1	Year 2	Year 3	Year 4	Year 5
B_0 0%	x1	x2	x3	x4	x5
B_1 2%	x6				
B_2 4%					x7
B_3 6%		x8			
B_4 6%		x9			

Decision Variables

Represent your decision variables by a vector x, where x(k) is the dollar amount of investment in bond k, for k=1..9. Upon maturity, the payout for investment x(k) is

$$x(k)(1 + \rho_k/100)^{m_k}.$$

Define r_k as the total return of bond k:

$$r_k = (1 + \rho_k/100)^{m_k}.$$

```
% Total returns
finalReturns = (1+InterestRates/100).^Maturity;
```

Objective Function

The goal is to choose investments to maximize the amount of money collected at the end of year T. From the plot, you see that investments are collected at various intermediate years and reinvested. At the end of year T, the money returned from investments 5, 7, and 8 can be collected and represents your final wealth:

$$\max_x x_5 r_5 + x_7 r_7 + x_8 r_8$$

To place this problem into the form linprog solves, turn this maximization problem into a minimization problem using the negative of the coefficients of x(j):

$$\min_{x} f^T x$$

with

$$f = [0; 0; 0; 0; -r_5; 0; -r_7; -r_8; 0]$$

```
f = zeros(nPtotal,1);
f([5,7,8]) = [-finalReturns(5),-finalReturns(7),-finalReturns(8)];
```

Linear Constraints: Invest No More Than You Have

Every year, you have a certain amount of money available to purchase bonds. Starting with year 1, you can invest the initial capital in the purchase options x_1 and x_6, so:

$$x_1 + x_6 = \text{Capital}_0$$

Then for the following years, you collect the returns from maturing bonds, and reinvest them in new available bonds to obtain the system of equations:

$$
\begin{aligned}
x_2 + x_8 + x_9 &= r_1 x_1 \\
x_3 &= r_2 x_2 \\
x_4 &= r_3 x_3 \\
x_5 + x_7 &= r_4 x_4 + r_6 x_6 + r_9 x_9
\end{aligned}
$$

Write these equations in the form $Aeq\,x = beq$, where each row of the Aeq matrix corresponds to the equality that needs to be satisfied that year:

$$
Aeq = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
-r_1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & -r_2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -r_3 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -r_4 & 1 & -r_6 & 1 & 0 & -r_9
\end{bmatrix}
$$

$$
beq = \begin{bmatrix}
\text{Capital}_0 \\
0 \\
0 \\
0
\end{bmatrix}
$$

```
Aeq = spalloc(N+1,nPtotal,15);
Aeq(1,[1,6]) = 1;
Aeq(2,[1,2,8,9]) = [-1,1,1,1];
Aeq(3,[2,3]) = [-1,1];
Aeq(4,[3,4]) = [-1,1];
Aeq(5,[4:7,9]) = [-finalReturns(4),1,-finalReturns(6),1,-finalReturns(9)];

beq = zeros(T,1);
beq(1) = Capital_0;
```

Bound Constraints: No Borrowing

Because each amount invested must be positive, each entry in the solution vector x must be positive. Include this constraint by setting a lower bound lb on the solution vector x. There is no explicit upper bound on the solution vector. Thus, set the upper bound ub to empty.

```
lb = zeros(size(f));
ub = [];
```

Solve the Problem

Solve this problem with no constraints on the amount you can invest in a bond. The interior-point algorithm can be used to solve this type of linear programming problem.

```
options = optimoptions('linprog','Algorithm','interior-point');
[xsol,fval,exitflag] = linprog(f,[],[],Aeq,beq,lb,ub,[],options);
```

```
Solution found during presolve.
```

Visualize the Solution

The exit flag is 1, indicating that the solver found a solution. The value `-fval`, returned as the second `linprog` output argument, corresponds to the final wealth. Plot your investments over time.

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);

plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol)
```

```
After 5 years, the return for the initial $1000 is $1262.48
```

	Year 1	Year 2	Year 3	Year 4	Year 5
B_0 0%	x1 = 1000.00	x2 = 0.00	x3 = 0.00	x4 = 0.00	x5 = 0.00
B_1 2%		x6 = 0.00			
B_2 4%					x7 = 0.00
B_3 6%		x8 = 1000.00			
B_4 6%		x9 = 0.00			

Optimal Investment with Limited Holdings

To diversify your investments, you can choose to limit the amount invested in any one bond to a certain percentage `Pmax` of the total capital that year (including the returns for bonds that are currently in their maturity period). You obtain the following system of inequalities:

$$
\begin{aligned}
x_1 &\leq \text{Pmax} \times \text{Capital}_0 \\
x_2 &\leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6) \\
x_3 &\leq \text{Pmax} \times (\rho_2 x_2 + \rho_6^2 x_6 + \rho_8 x_8 + \rho_9 x_9) \\
x_4 &\leq \text{Pmax} \times (\rho_3 x_3 + \rho_6^3 x_6 + \rho_8^2 x_8 + \rho_9^2 x_9) \\
x_5 &\leq \text{Pmax} \times (\rho_4 x_4 + \rho_6^4 x_4 + \rho_8^3 x_8 + \rho_9^3 x_9) \\
x_6 &\leq \text{Pmax} \times \text{Capital}_0 \\
x_7 &\leq \text{Pmax} \times (\rho_4 x_4 + \rho_6^4 x_4 + \rho_8^3 x_8 + \rho_9^3 x_9) \\
x_8 &\leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6) \\
x_9 &\leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6)
\end{aligned}
$$

Place these inequalities in the matrix form `Ax <= b`.

To set up the system of inequalities, first generate a matrix `yearlyReturns` that contains the return for the bond indexed by i at year j in row i and column j. Represent this system as a sparse matrix.

```
% Maximum percentage to invest in any bond
```

```
Pmax = 0.6;

% Build the return for each bond over the maturity period as a sparse
% matrix
cumMaturity = [0;cumsum(Maturity)];
xr = zeros(cumMaturity(end-1),1);
yr = zeros(cumMaturity(end-1),1);
cr = zeros(cumMaturity(end-1),1);
for i = 1:nPtotal
    mi = Maturity(i); % maturity of bond i
    pi = PurchaseYears(i); % purchase year of bond i
    idx = cumMaturity(i)+1:cumMaturity(i+1); % index into xr, yr and cr
    xr(idx) = i; % bond index
    yr(idx) = pi+1:pi+mi; % maturing years
    cr(idx) = (1+InterestRates(i)/100).^(1:mi); % returns over the maturity period
end
yearlyReturns = sparse(xr,yr,cr,nPtotal,T+1);

% Build the system of inequality constraints
A = -Pmax*yearlyReturns(:,PurchaseYears)'+ speye(nPtotal);

% Left-hand side
b = zeros(nPtotal,1);
b(PurchaseYears == 1) = Pmax*Capital_0;
```

Solve the problem by investing no more than 60% in any one asset. Plot the resulting purchases. Notice that your final wealth is less than the investment without this constraint.

```
[xsol,fval,exitflag] = linprog(f,A,b,Aeq,beq,lb,ub,[],options);

fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);

plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol)


Minimum found that satisfies the constraints.

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

After 5 years, the return for the initial $1000 is $1207.78
```

Model of Arbitrary Size

Create a model for a general version of the problem. Illustrate it using T = 30 years and 400 randomly generated bonds with interest rates from 1 to 6%. This setup results in a linear programming problem with 430 decision variables. The system of equality constraints is represented by a sparse matrix Aeq of dimension 30-by-430 and the system of inequalities is represented by a sparse matrix A of dimension 430-by-430.

```
% for reproducibility
rng default
% Initial amount of money
Capital_0 = 1000;
% Time period in years
T = 30;
% Number of bonds
N = 400;
% Total number of buying oportunities
nPtotal = N+T;
% Generate random maturity durations
Maturity = randi([1 T-1],nPtotal,1);
% Bond 1 has a maturity period of 1 year
Maturity(1:T) = 1;
% Generate random yearly interest rate for each bond
InterestRates = randi(6,nPtotal,1);
% Bond 1 has an interest rate of 0 (not invested)
InterestRates(1:T) = 0;
% Compute the return at the end of the maturity period for each bond:
finalReturns = (1+InterestRates/100).^Maturity;
```

```
% Generate random purchase years for each option
PurchaseYears = zeros(nPtotal,1);
% Bond 1 is available for purchase every year
PurchaseYears(1:T)=1:T;
for i=1:N
    % Generate a random year for the bond to mature before the end of
    % the T year period
    PurchaseYears(i+T) = randi([1 T-Maturity(i+T)+1]);
end

% Compute the years where each bond reaches maturity
SaleYears = PurchaseYears + Maturity;

% Initialize f to 0
f = zeros(nPtotal,1);
% Indices of the sale oportunities at the end of year T
SalesTidx = SaleYears==T+1;
% Expected return for the sale oportunities at the end of year T
ReturnsT = finalReturns(SalesTidx);
% Objective function
f(SalesTidx) = -ReturnsT;


% Generate the system of equality constraints.
% For each purchase option, put a coefficient of 1 in the row corresponding
% to the year for the purchase option and the column corresponding to the
% index of the purchase oportunity
xeq1 = PurchaseYears;
yeq1 = (1:nPtotal)';
ceq1 = ones(nPtotal,1);

% For each sale option, put -\rho_k, where \rho_k is the interest rate for the
% associated bond that is being sold, in the row corresponding to the
% year for the sale option and the column corresponding to the purchase
% oportunity
xeq2 = SaleYears(~SalesTidx);
yeq2 = find(~SalesTidx);
ceq2 = -finalReturns(~SalesTidx);

% Generate the sparse equality matrix
Aeq = sparse([xeq1; xeq2], [yeq1; yeq2], [ceq1; ceq2], T, nPtotal);

% Generate the right-hand side
```

```
beq = zeros(T,1);
beq(1) = Capital_0;

% Build the system of inequality constraints
% Maximum percentage to invest in any bond
Pmax = 0.4;

% Build the returns for each bond over the maturity period
cumMaturity = [0;cumsum(Maturity)];
xr = zeros(cumMaturity(end-1),1);
yr = zeros(cumMaturity(end-1),1);
cr = zeros(cumMaturity(end-1),1);
for i = 1:nPtotal
    mi = Maturity(i); % maturity of bond i
    pi = PurchaseYears(i); % purchase year of bond i
    idx = cumMaturity(i)+1:cumMaturity(i+1); % index into xr, yr and cr
    xr(idx) = i; % bond index
    yr(idx) = pi+1:pi+mi; % maturing years
    cr(idx) = (1+InterestRates(i)/100).^(1:mi); % returns over the maturity period
end
yearlyReturns = sparse(xr,yr,cr,nPtotal,T+1);

% Build the system of inequality constraints
A = -Pmax*yearlyReturns(:,PurchaseYears)'+ speye(nPtotal);

% Left-hand side
b = zeros(nPtotal,1);
b(PurchaseYears==1) = Pmax*Capital_0;

% Add the lower-bound constraints to the problem.
lb = zeros(nPtotal,1);
```

Solution with No Holding Limit

First, solve the linear programming problem without inequality constraints using the interior-point algorithm.

```
% Solve the problem without inequality constraints
options = optimoptions('linprog','Algorithm','interior-point');
tic
[xsol,fval,exitflag] = linprog(f,[],[],Aeq,beq,lb,[],[],options);
toc

fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
Minimum found that satisfies the constraints.

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

Elapsed time is 0.013190 seconds.

After 30 years, the return for the initial $1000 is $5167.58
```

Solution with Limited Holdings

Now, solve the problem with the inequality constraints.

```
% Solve the problem with inequality constraints
options = optimoptions('linprog','Algorithm','interior-point');
tic
[xsol,fval,exitflag] = linprog(f,A,b,Aeq,beq,lb,[],[],options);
toc

fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
Minimum found that satisfies the constraints.

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

Elapsed time is 1.235519 seconds.

After 30 years, the return for the initial $1000 is $5095.26
```

Even though the number of constraints increased by an order of 10, the time for the
solver to find a solution increased by an order of 100. This performance discrepancy is
partially caused by dense columns in the inequality system shown in matrix A. These
columns correspond to bonds with a long maturity period, as shown in the following
graph.

```
% Number of nonzero elements per column
nnzCol = sum(spones(A));
```

```
% Plot the maturity length vs. the number of nonzero elements for each bond
figure;
plot(Maturity,nnzCol,'o');
xlabel('Maturity period of bond k')
ylabel('Number of nonzero in column k of A')
```
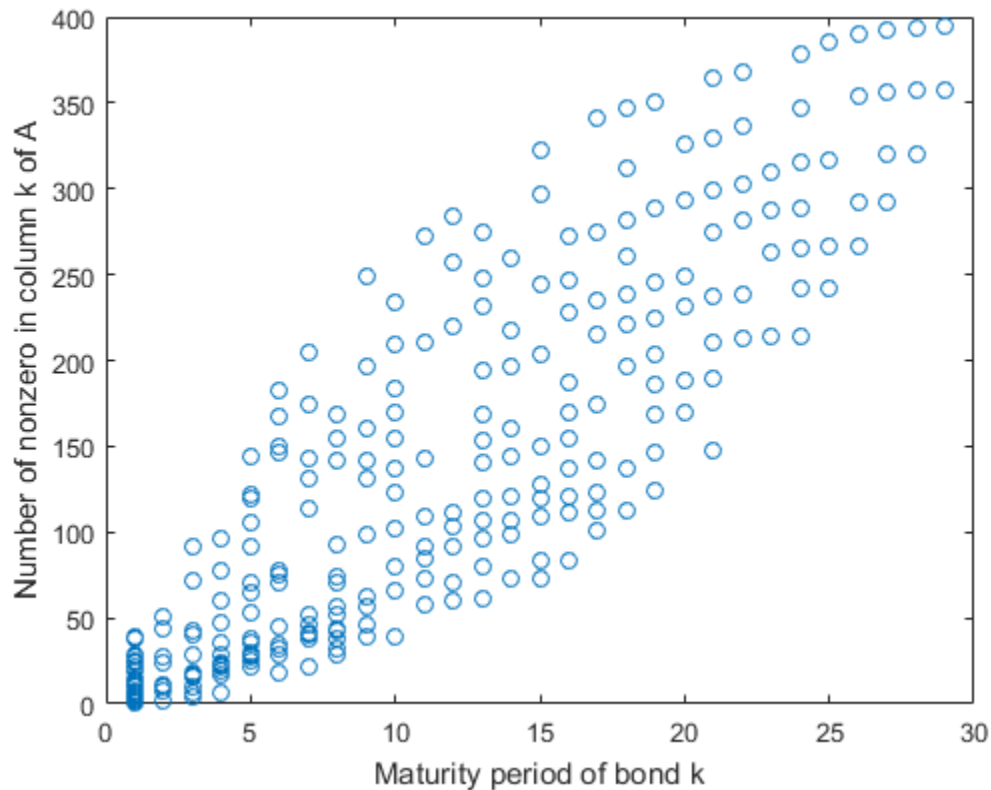

Dense columns in the constraints lead to dense blocks in the solver's internal matrices, yielding a loss of efficiency of its sparse methods. To speed up the solver, try the dual-simplex algorithm, which is less sensitive to column density.

```
% Solve the problem with inequality constraints using dual simplex
options = optimoptions('linprog','Algorithm','dual-simplex');
tic
```

```
[xsol,fval,exitflag] = linprog(f,A,b,Aeq,beq,lb,[],[],options);
toc

fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
Optimal solution found.

Elapsed time is 0.254719 seconds.

After 30 years, the return for the initial $1000 is $5095.26
```

In this case, the dual-simplex algorithm took much less time to obtain the same solution.

Qualitative Result Analysis

To get a feel for the solution found by `linprog`, compare it to the amount `fmax` that you would get if you could invest all of your starting money in one bond with a 6% interest rate (the maximum interest rate) over the full 30 year period. You can also compute the equivalent interest rate corresponding to your final wealth.

```
% Maximum amount
fmax = Capital_0*(1+6/100)^T;
% Ratio (in percent)
rat = -fval/fmax*100;
% Equivalent interest rate (in percent)
rsol = ((-fval/Capital_0)^(1/T)-1)*100;

fprintf(['The amount collected is %g%% of the maximum amount $%g '...
    'that you would obtain from investing in one bond.\n'...
    'Your final wealth corresponds to a %g%% interest rate over the %d year '...
    'period.\n'], rat, fmax, rsol, T)

plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol,false)
```

```
The amount collected is 88.7137% of the maximum amount $5743.49 that you would obtain f
Your final wealth corresponds to a 5.57771% interest rate over the 30 year period.
```
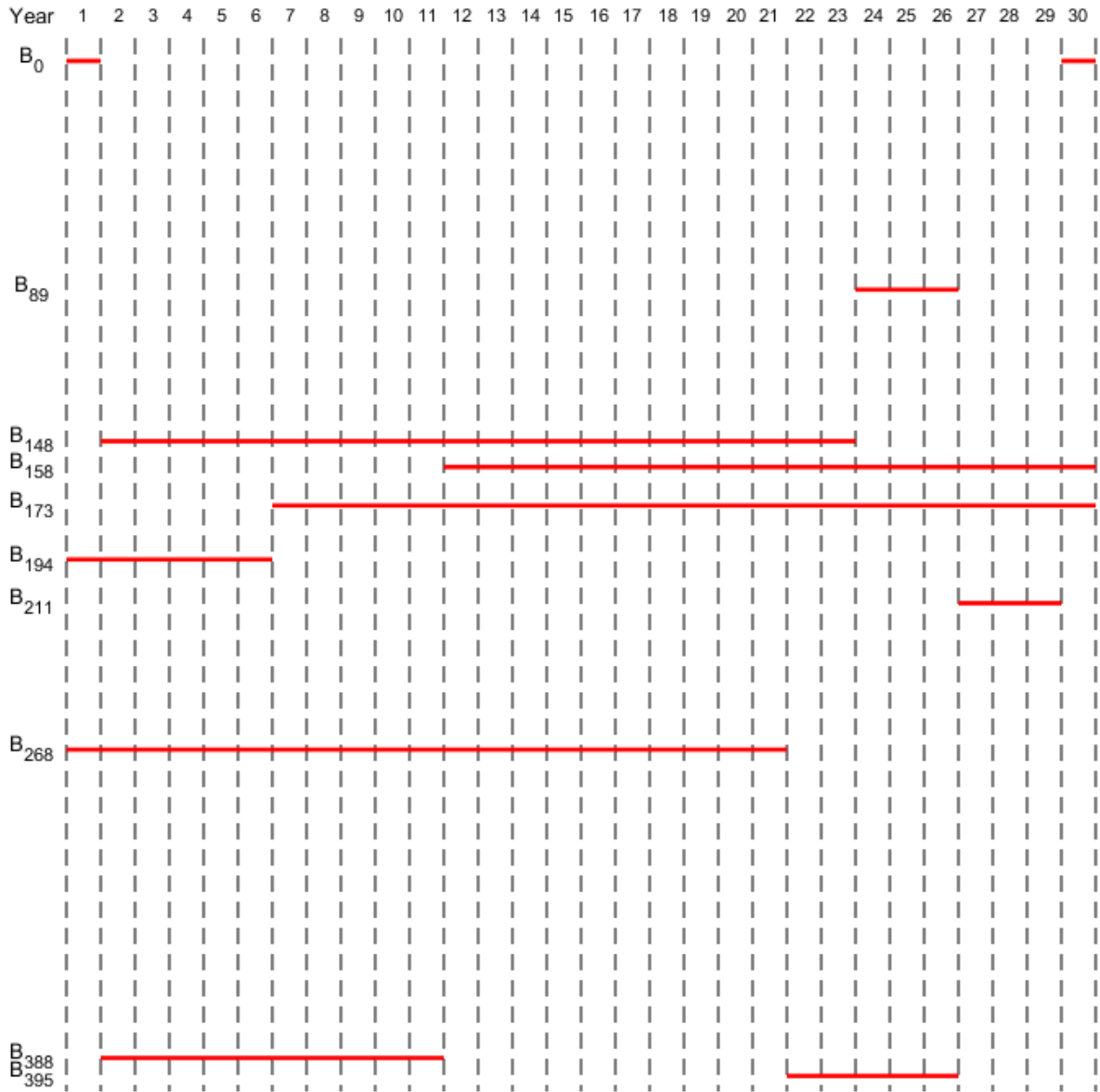
Mixed-Integer Linear Programming Algorithms

In this section...
"Mixed-Integer Linear Programming Definition" on page 8-38
"intlinprog Algorithm" on page 8-38

Mixed-Integer Linear Programming Definition

A mixed-integer linear program is a problem with

- Linear objective function, $f^{\mathrm{T}}x$, where f is a column vector of constants, and x is the column vector of unknowns

- Bounds and linear constraints, but no nonlinear constraints (for definitions, see "Writing Constraints" on page 2-31)

- Restrictions on some components of x to have integer values

In mathematical terms, given vectors f, lb, and ub, matrices A and Aeq, corresponding vectors b and beq, and a set of indices $\texttt{intcon}$, find a vector x to solve

$$\min_x f^T x \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub. \end{cases}$$

intlinprog Algorithm

- "Algorithm Overview" on page 8-39
- "Linear Program Preprocessing" on page 8-39
- "Linear Programming" on page 8-40
- "Mixed-Integer Program Preprocessing" on page 8-40
- "Cut Generation" on page 8-40
- "Heuristics for Finding Feasible Solutions" on page 8-41
- "Branch and Bound" on page 8-42

Algorithm Overview

`intlinprog` uses this basic strategy to solve mixed-integer linear programs. `intlinprog` can solve the problem in any of the stages. If it solves the problem in a stage, `intlinprog` does not execute the later stages.

1 Reduce the problem size using "Linear Program Preprocessing" on page 8-39.
2 Solve an initial relaxed (noninteger) problem using "Linear Programming" on page 8-40.
3 Perform "Mixed-Integer Program Preprocessing" on page 8-40 to tighten the LP relaxation of the mixed-integer problem.
4 Try "Cut Generation" on page 8-40 to further tighten the LP relaxation of the mixed-integer problem.
5 Try to find integer-feasible solutions using heuristics.
6 Use a "Branch and Bound" on page 8-42 algorithm to search systematically for the optimal solution. This algorithm solves LP relaxations with restricted ranges of possible values of the integer variables. It attempts to generate a sequence of updated bounds on the optimal objective function value.

Linear Program Preprocessing

According to the "Mixed-Integer Linear Programming Definition" on page 8-38, there are matrices *A* and *Aeq* and corresponding vectors *b* and *beq* that encode a set of linear inequalities and linear equalities

$$A \cdot x \le b$$
$$Aeq \cdot x = beq.$$

These linear constraints restrict the solution *x*.

Usually, it is possible to reduce the number of variables in the problem (the number of components of *x*), and reduce the number of linear constraints. While performing these reductions can take time for the solver, they usually lower the overall time to solution, and can make larger problems solvable. The algorithms can make solution more numerically stable. Furthermore, these algorithms can sometimes detect an infeasible problem.

Preprocessing steps aim to eliminate redundant variables and constraints, improve the scaling of the model and sparsity of the constraint matrix, strengthen the bounds on variables, and detect the primal and dual infeasibility of the model.

For details, see Andersen and Andersen [1] and Mészáros and Suhl [4].

Linear Programming

The initial *relaxed* problem is the linear programming problem with the same objective and constraints as "Mixed-Integer Linear Programming Definition" on page 8-38, but no integer constraints. Call x_{LP} the solution to the relaxed problem, and x the solution to the original problem with integer constraints. Clearly,

$f^{\mathrm{T}}x_{\mathrm{LP}} \le f^{\mathrm{T}}x,$

because x_{LP} minimizes the same function but with fewer restrictions.

This initial relaxed LP (root node LP) and all generated LP relaxations during the branch-and-bound algorithm are solved using linear programming solution techniques.

Mixed-Integer Program Preprocessing

During mixed-integer program preprocessing, `intlinprog` analyzes the linear inequalities `A*x ≤ b` along with integrality restrictions to determine whether:

- The problem is infeasible.
- Some bounds can be tightened.
- Some inequalities are redundant, so can be ignored or removed.
- Some inequalities can be strengthened.
- Some integer variables can be fixed.

The `IPPreprocess` option lets you choose whether `intlinprog` takes several steps, takes all of them, or takes almost none of them.

The main goal of mixed-integer program preprocessing is to simplify ensuing branch-and-bound calculations. Preprocessing involves quickly preexamining and eliminating some of the futile subproblem candidates that branch-and-bound would otherwise analyze.

For details about integer preprocessing, see Savelsbergh [6].

Cut Generation

Cuts are additional linear inequality constraints that `intlinprog` adds to the problem. These inequalities attempt to restrict the feasible region of the LP relaxations so that their solution are closer to integers. You control the type of cuts that `intlinprog` uses with the `CutGeneration` option.

'basic' cuts include:

- Mixed-integer rounding cuts
- Gomory cuts
- Cliques cuts
- Cover cuts
- Flow cover cuts

'intermediate' cuts include all 'basic' cuts, plus:

- Simple lift-and-project cuts
- Simple pivot-and-reduce cuts
- Reduce-and-split cuts

'advanced' cuts include all 'intermediate' cuts except reduce-and-split cuts, plus:

- Strong Chvatal-Gomory cuts
- Zero-half cuts

Another option, CutGenMaxIter, specifies an upper bound on the number of times intlinprog iterates to generate cuts.

For details about cut generation algorithms (also called cutting plane methods), see Cornuéjols [2].

Heuristics for Finding Feasible Solutions

To get an upper bound on the objective function, the branch-and-bound procedure must find feasible points. A solution to an LP relaxation during branch-and-bound can be integer feasible, which can provide an improved upper bound to the original MILP. There are techniques for finding feasible points faster before and/or during branch-and-bound. These techniques are heuristic, meaning they are algorithms that can succeed, but can also fail. You set the intlinprog heuristics in the Heuristics option. The options are:

- 'rins' — intlinprog searches the neighborhood of the current best integer feasible solution point (if available) to find a new and better solution. See Danna, Rothberg, and Le Pape [3].
- 'rss' — intlinprog applies a hybrid procedure combining ideas from 'rins' and local branching to search for integer feasible solutions.
- 'round' — intlinprog takes the LP solution to the relaxed problem at a node. It rounds the integer components in a way that attempts to maintain feasibility.

- 'none' — `intlinprog` does not search for a feasible point. It simply takes any feasible point it encounters in its branch-and-bound search.

Branch and Bound

The branch-and-bound method constructs a sequence of subproblems that attempt to converge to a solution of the MILP. The subproblems give a sequence of upper and lower bounds on the solution $f^T x$. The first upper bound is any feasible solution, and the first lower bound is the solution to the relaxed problem. For a discussion of the upper bound, see "Heuristics for Finding Feasible Solutions" on page 8-41.

As explained in "Linear Programming" on page 8-40, any solution to the linear programming relaxed problem has a lower objective function value than the solution to the MILP. Also, any feasible point x_{feas} satisfies

$$f^T x_{\text{feas}} \geq f^T x,$$

because $f^T x$ is the minimum among all feasible points.

In this context, a *node* is an LP with the same objective function, bounds, and linear constraints as the original problem, but without integer constraints, and with particular changes to the linear constraints or bounds. The *root node* is the original problem with no integer constraints and no changes to the linear constraints or bounds, meaning the root node is the initial relaxed LP.

From the starting bounds, the branch-and-bound method constructs new subproblems by branching from the root node. The branching step is taken heuristically, according to one of several rules. Each rule is based on the idea of splitting a problem by restricting one variable to be less than or equal to an integer J, or greater than or equal to J+1. These two subproblems arise when an entry in x_{LP}, corresponding to an integer specified in intcon, is not an integer. Here, x_{LP} is the solution to a relaxed problem. Take J as the floor of the variable (rounded down), and J+1 as the ceiling (rounded up). The resulting two problems have solutions that are larger than or equal to $f^T x_{\text{LP}}$, because they have more restrictions. Therefore, this procedure potentially raises the lower bound.

The performance of the branch-and-bound method depends on the rule for choosing which variable to split (the branching rule). The algorithm uses these rules, which you can set in the `BranchingRule` option:

- 'maxpscost' — Choose the fractional variable with maximal *pseudocost*.

Pseudocost

The peudocost of a variable i is based on empirical estimates of the change in the lower bound when i has been chosen as the branching variable, combined with the fractional part of the i component of the current point x. The fractional part p is in two pieces, the lower part and the upper part:

$p_i^- = x(i) - \lfloor x(i) \rfloor$

$p_i^+ = 1 - p_i^-$.

Let x_i^- be the solution of the linear program restricted to have $x(i) \le \lfloor x(i) \rfloor$, and let the change in objective function be denoted

$\Delta_i^- = f^T x_i^- - f^T x$.

Similarly, Δ_i^+ is the change in objective function when the problem is restricted to have $x(i) \ge \lceil x(i) \rceil$.

The objective gain per unit change in variable x_i is

$$d_i^- = \frac{\Delta_i^-}{p_i^-} \text{ or } d_i^+ = \frac{\Delta_i^+}{p_i^+}.$$

Let s_i^- and s_i^+ be the empirical averages of d_i^- and d_i^+ during the branch-and-bound algorithm up to this point. The empirical values are initialized to the absolute value of the objective coefficient $f(i)$ for the terms before there are any observations. Then the 'maxpscost' rule is to branch on a node i that maximizes, for some positive weights w^+ and w^-, the quantity

$w^- * p_i^- * s_i^- + w^+ * p_i^+ * s_i^+$.

Roughly speaking, this rule chooses a coefficient that is likely to increase the lower bound maximally.

- 'mostfractional' — Choose the variable with fractional part closest to $1/2$.
- 'maxfun' — Choose the variable with maximal corresponding absolute value in the objective vector f.

After the algorithm branches, there are two new nodes to explore. The algorithm chooses which node to explore among all that are available using one of these rules:

- `'minobj'` — Choose the node that has the lowest objective function value.
- `'mininfeas'` — Choose the node with the minimal sum of integer infeasibilities. This means for every integer-infeasible component $x(i)$ in the node, add up the smaller of p_i^- and p_i^+, where

  $p_i^- = x(i) - \lfloor x(i) \rfloor$

  $p_i^+ = 1 - p_i^-$.

- `'simplebestproj'` — Choose the node with the *best projection*.

  Best Projection

  Let x_B denote the best integer-feasible point found so far, x_R demote the LP relaxed solution at the root node, and x denote the node we examine. Let $in(x)$ denote the sum of integer infeasibilities at the node x (see `'mininfeas'`). The best projection rule is to minimize

  $$f^T x + \frac{f^T x_B - f^T x_R}{in(x_R)} in(x).$$

  If there is no integer-feasible point found so far, set $f^T x_B = 0$.

The branch-and-bound procedure continues, systematically generating subproblems to analyze and discarding the ones that won't improve an upper or lower bound on the objective, until one of these stopping criteria is met:

- The algorithm exceeds the `MaxTime` option.
- The difference between the lower and upper bounds on the objective function is less than the `TolGapAbs` or `TolGapRel` tolerances.
- The number of explored nodes exceeds the `MaxNodes` option.
- The number of integer feasible points exceeds the `MaxNumFeasPoints` option.

For details about the branch-and-bound procedure, see Nemhauser and Wolsey [5] and Wolsey [7].

References

[1] Andersen, E. D., and Andersen, K. D. *Presolving in linear programming.* Mathematical Programming 71, pp. 221–245, 1995.

[2] Cornuéjols, G. *Valid inequalities for mixed integer linear programs.* Mathematical Programming B, Vol. 112, pp. 3–44, 2008.

[3] Danna, E., Rothberg, E., Le Pape, C. *Exploring relaxation induced neighborhoods to improve MIP solutions.* Mathematical Programming, Vol. 102, issue 1, pp. 71–90, 2005.

[4] Mészáros C., and Suhl, U. H. *Advanced preprocessing techniques for linear and quadratic programming.* OR Spectrum, 25(4), pp. 575–595, 2003.

[5] Nemhauser, G. L. and Wolsey, L. A. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, 1999.

[6] Savelsbergh, M. W. P. *Preprocessing and Probing Techniques for Mixed Integer Programming Problems.* ORSA J. Computing, Vol. 6, No. 4, pp. 445–454, 1994.

[7] Wolsey, L. A. *Integer Programming.* Wiley-Interscience, New York, 1998.

Tuning Integer Linear Programming

In this section...
"Change Options to Improve the Solution Process" on page 8-46
"Some "Integer" Solutions Are Not Integers" on page 8-47
"Large Components Not Integer Valued" on page 8-47
"Large Coefficients Disallowed" on page 8-48

Change Options to Improve the Solution Process

Note: Often, you can change the formulation of a MILP to make it more easily solvable. For suggestions on how to change your formulation, see Williams [1].

After you run `intlinprog` once, you might want to change some options and rerun it. The changes you might want to see include:

- Lower run time
- Lower final objective function value (a better solution)
- Smaller final gap
- More or different feasible points

Here are general recommendations for option changes that are most likely to help the solution process. Try the suggestions in this order:

1 For a faster and more accurate solution, increase the `CutGenMaxIter` option from its default `10` to a higher number such as `25`. This can speed up the solution, but can also slow it.

2 For a faster and more accurate solution, change the `CutGeneration` option to `'intermediate'` or `'advanced'`. This can speed up the solution, but can use much more memory, and can slow the solution.

3 For a faster and more accurate solution, change the `IPPreprocess` option to `'advanced'`. This can have a large effect on the solution process, either beneficial or not.

4 For a faster and more accurate solution, change the `RootLPAlgorithm` option to `'primal-simplex'`. Usually this change is not beneficial, but occasionally it can be.

5 To try to find more or better feasible points, increase the `HeuristicsMaxNodes` option from its default `50` to a higher number such as `100`.

6 To try to find more or better feasible points, change the `Heuristics` option to either `'round'` or `'rins'`.

7 To attempt to stop the solver more quickly, change the `TolGapRel` option to a higher value than the default `1e-4`. Similarly, to attempt to obtain a more accurate answer, change the `TolGapRel` option to a lower value. These changes do not always improve results.

8 For a more accurate solution, decrease the `RelObjThreshold` option from its default `1e-4` to a smaller positive value such as `1e-6`. This change can cause `intlinprog` to take more time to solve the problem, and to find more integer feasible points during its solution process.

Some "Integer" Solutions Are Not Integers

Often, some supposedly integer-valued components of the solution `x(intcon)` are not precisely integers. `intlinprog` considers as integers all solution values within the `TolInteger` tolerance of an integer.

To round all supposed integers to be precisely integers, use the `round` function.

```
x(intcon) = round(x(intcon));
```

Caution Rounding can cause solutions to become infeasible. Check feasibility after rounding:

```
max(A*x - b) % see if entries are not too positive, so have small infeasibility
max(abs(Aeq*x - beq)) % see if entries are near enough to zero
max(x - ub) % positive entries are violated bounds
max(lb - x) % positive entries are violated bounds
```

Large Components Not Integer Valued

`intlinprog` does not enforce that solution components be integer valued when their absolute values exceed `2.1e9`. When your solution has such components, `intlinprog` warns you. If you receive this warning, check the solution to see whether supposedly integer-valued components of the solution are close to integers.

Large Coefficients Disallowed

`intlinprog` does not allow components of the problem, such as coefficients in `f`, `A`, or `ub`, to exceed `1e15` in absolute value. If you try to run `intlinprog` with such a problem, `intlinprog` issues an error.

If you get this error, sometimes you can scale the problem to have smaller coefficients:

- For coefficients in `f` that are too large, try multiplying `f` by a small positive scaling factor.
- For constraint coefficients that are too large, try multiplying all bounds and constraint matrices by the same small positive scaling factor.

References

[1] Williams, H. Paul. *Model Building in Mathematical Programming.* Wiley, 2013.

Mixed-Integer Linear Programming Basics

This example shows how to solve a mixed-integer linear program. The example is not complex, but it shows typical steps in formulating a problem in the syntax for `intlinprog`.

Problem description

You want to blend a variety of steels with various chemical compositions to obtain 25 tons of steel with a specific chemical composition. The result should have 5% carbon and 5% molybdenum by weight, meaning 25 tons*5% = 1.25 tons of carbon and 1.25 tons of molybdenum. The objective is to minimize the cost for blending the steel.

This problem is taken from Carl-Henrik Westerberg, Bengt Bjorklund and Eskil Hultman, "*An Application of Mixed Integer Programming in a Swedish Steel Mill.*" Interfaces February 1977 Vol. 7, No. 2 pp. 39–43, whose abstract is at `http://interfaces.journal.informs.org/content/7/2/39.abstract`.

Four ingots of steel are available for purchase. Only one of each ingot is available.

Ingot	Weight (tons)	%Carbon	%Molybdenum	Cost/ton
1	5	5	3	$350
2	3	4	3	$330
3	4	5	4	$310
4	6	3	4	$280

Three grades of alloy steel are available for purchase, and one grade of scrap steel. Alloy and scrap steels can be purchased in fractional amounts.

Alloy	%Carbon	%Molybdenum	Cost/ton
1	8	6	$500
2	7	7	$450
3	6	8	$400
Scrap	3	9	$100

To formulate the problem, first decide on the control variables. Take variable `x(1) = 1` to mean you purchase ingot **1**, and `x(1) = 0` to mean you do not purchase the ingot.

Similarly, variables `x(2)` through `x(4)` are binary variables indicating that you purchase ingots **2** through **4**.

Variables `x(5)` through `x(7)` are the quantities of alloys **1**, **2**, and **3** you purchase, and `x(8)` is the quantity of scrap steel you purchase.

MATLAB formulation

Formulate the problem by specifying the inputs for `intlinprog`. The relevant `intlinprog` syntax is as follows.

```
[x,fval] = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)
```

Create the inputs for `intlinprog` from first (`f`) through last (`ub`).

`f` is the vector of cost coefficients. The coefficients representing the costs of ingots are the ingot weights times their cost per ton.

```
f = [350*5,330*3,310*4,280*6,500,450,400,100];
```

The integer variables are the first four.

```
intcon = 1:4;
```

Tip To specify binary variables, set the variables to be integers in `intcon`, and give them a lower bound of `0` and an upper bound of `1`.

There are no linear inequality constraints, so `A` and `b` are empty `[]`.

There are three equality constraints. The first is that the total weight is 25 tons.
`5*x(1) + 3*x(2) + 4*x(3) + 6*x(4) + x(5) + x(6) + x(7) + x(8) = 25`.

The second constraint is that the weight of carbon is 5% of 25 tons, or 1.25 tons.
`5*0.05*x(1) + 3*0.04*x(2) + 4*0.05*x(3) + 6*0.03*x(4)`
`+ 0.08*x(5) + 0.07*x(6) + 0.06*x(7) + 0.03*x(8) = 1.25`.

The third constraint is that the weight of molybdenum is 1.25 tons.
`5*0.03*x(1) + 3*0.03*x(2) + 4*0.04*x(3) + 6*0.04*x(4)`
`+ 0.06*x(5) + 0.07*x(6) + 0.08*x(7) + 0.09*x(8) = 1.25`.

In matrix form, `Aeq*x = beq`, where

```
Aeq = [5,3,4,6,1,1,1,1;
    5*0.05,3*0.04,4*0.05,6*0.03,0.08,0.07,0.06,0.03;
    5*0.03,3*0.03,4*0.04,6*0.04,0.06,0.07,0.08,0.09];
beq = [25;1.25;1.25];
```

Each variable is bounded below by zero. The integer variables are bounded above by one.

```
lb = zeros(8,1);
ub = ones(8,1);
ub(5:end) = Inf; % No upper bound on noninteger variables
```

Solve the problem

Now that you have all the inputs, call the solver.

```
[x,fval] = intlinprog(f,intcon,[],[],Aeq,beq,lb,ub);
```

View the solution.

```
x,fval

x =

    1.0000
    1.0000
         0
    1.0000
    7.2500
         0
    0.2500
    3.5000


fval =

   8.4950e+03
```

The optimal purchase costs \$8,495. Buy ingots **1**, **2**, and **4**, but not **3**, and buy 7.25 tons of alloy **1**, 0.25 ton of alloy **3**, and 3.5 tons of scrap steel.

Set `intcon = []` to see the effect of solving the problem without integer constraints. The solution is different, and is not sensible, because you cannot purchase a fraction of an ingot.

Factory, Warehouse, Sales Allocation Model

This example shows how to set up and solve a mixed-integer linear programming problem. The problem is to find the optimal production and distribution levels among a set of factories, warehouses, and sales outlets.

The example first generates random locations for factories, warehouses, and sales outlets. Feel free to modify the scaling parameter N, which scales both the size of the grid in which the production and distribution facilities reside, but also scales the number of these facilities so that the density of facilities of each type per grid area is independent of N.

Facility Locations

For a given value of the scaling parameter N, suppose that there are the following:

- $\lfloor fN^2 \rfloor$ factories
- $\lfloor wN^2 \rfloor$ warehouses
- $\lfloor sN^2 \rfloor$ sales outlets

These facilities are on separate integer grid points between 1 and N in the x and y directions. In order that the facilities have separate locations, you require that $f + w + s \leq 1$. In this example, take $N = 20$, $f = 0.05$, $w = 0.05$, and $s = 0.1$.

Production and Distribution

There are P products made by the factories. Take $P = 20$.

The demand for each product p in a sales outlet s is $d(s, p)$. The demand is the quantity that can be sold in a time interval. One constraint on the model is that the demand is met, meaning the system produces and distributes exactly the quantities in the demand.

There are capacity constraints on each factory and each warehouse.

- The production of product p at factory f is less than $pcap(f, p)$.
- The capacity of warehouse w is $wcap(w)$.
- The amount of product p that can be transported from warehouse w to a sales outlet in the time interval is less than $turn(p) * wcap(w)$, where $turn(p)$ is the turnover rate of product p.

Suppose that each sales outlet receives its supplies from just one warehouse. Part of the problem is to determine the cheapest mapping of sales outlets to warehouses.

Costs

The cost of transporting products from factory to warehouse, and from warehouse to sales outlet, depends on the distance between the facilities, and on the particular product. If $dist(a, b)$ is the distance between facilities a and b, then the cost of shipping a product p between these facilities is the distance times the transportation cost $tcost(p)$:

$$dist(a, b) * tcost(p).$$

The distance in this example is the grid distance, also known as the L_1 distance. It is the sum of the absolute difference in x coordinates and y coordinates.

The cost of making a unit of product p in factory f is $pcost(f, p)$.

Optimization Problem

Given a set of facility locations, and the demands and capacity constraints, find:

- A production level of each product at each factory
- A distribution schedule for products from factories to warehouses
- A distribution schedule for products from warehouses to sales outlets

These quantities must ensure that demand is satisfied and total cost is minimized. Also, each sales outlet is required to receive all its products from exactly one warehouse.

Variables and Equations for the Optimization Problem

The control variables, meaning the ones you can change in the optimization, are

- $x(p, f, w)$ = the amount of product p that is transported from factory f to warehouse w

- $y(s, w)$ = a binary variable taking value 1 when sales outlet s is associated with warehouse w

The objective function to minimize is

$$\sum_f \sum_p \sum_w x(p, f, w) \ \cdot \ (pcost(f, p) + tcost(p) \cdot dist(f, w))$$

$$+\sum_s \sum_w \sum_p (d(s,p) \ \cdot \ tcost(p) \ \cdot \ dist(s,w) \ \cdot \ y(s,w)).$$

The constraints are

$\sum_w x(p,f,w) \le pcap(f,p)$ (capacity of factory).

$\sum_f x(p,f,w) = \sum_s (d(s,p) \ \cdot \ y(s,w))$ (demand is met).

$\sum_p \sum_s \frac{d(s,p)}{turn(p)} \ \cdot \ y(s,w) \le wcap(w)$ (capacity of warehouse).

$\sum_w y(s,w) = 1$ (each sales outlet associates to one warehouse).

$x(p,f,w) \ge 0$ (nonnegative production).

$y(s,w) \ \epsilon \ \{0,1\}$ (binary y).

The variables x and y appear in the objective and constraint functions linearly. Because y is restricted to integer values, the problem is a mixed-integer linear program (MILP).

Generate a Random Problem: Facility Locations

Set the values of the N, f, w, and s parameters, and generate the facility locations.

```
rng default % for reproducibility
N = 20; % N from 10 to 30 seems to work. Choose large values with caution.
N2 = N*N;
f = 0.05; % density of factories
w = 0.05; % density of warehouses
s = 0.1; % density of sales outlets

F = floor(f*N2); % number of factories
W = floor(w*N2); % number of warehouses
S = floor(s*N2); % number of sales outlets

xyloc = randperm(N2,F+W+S); % unique locations of facilities
[xloc,yloc] = ind2sub([N N],xyloc);
```

Of course, it is not realistic to take random locations for facilities. This example is intended to show solution techniques, not how to generate good facility locations.

Plot the facilities. Facilities 1 through F are factories, F+1 through F+W are warehouses, and F+W+1 through F+W+S are sales outlets.

```
h = figure;
plot(xloc(1:F),yloc(1:F),'rs',xloc(F+1:F+W),yloc(F+1:F+W),'k*',...
    xloc(F+W+1:F+W+S),yloc(F+W+1:F+W+S),'bo');
legend('Factory','Warehouse','Sales outlet','Location','EastOutside')
xlim([O N+1]);ylim([O N+1])
```
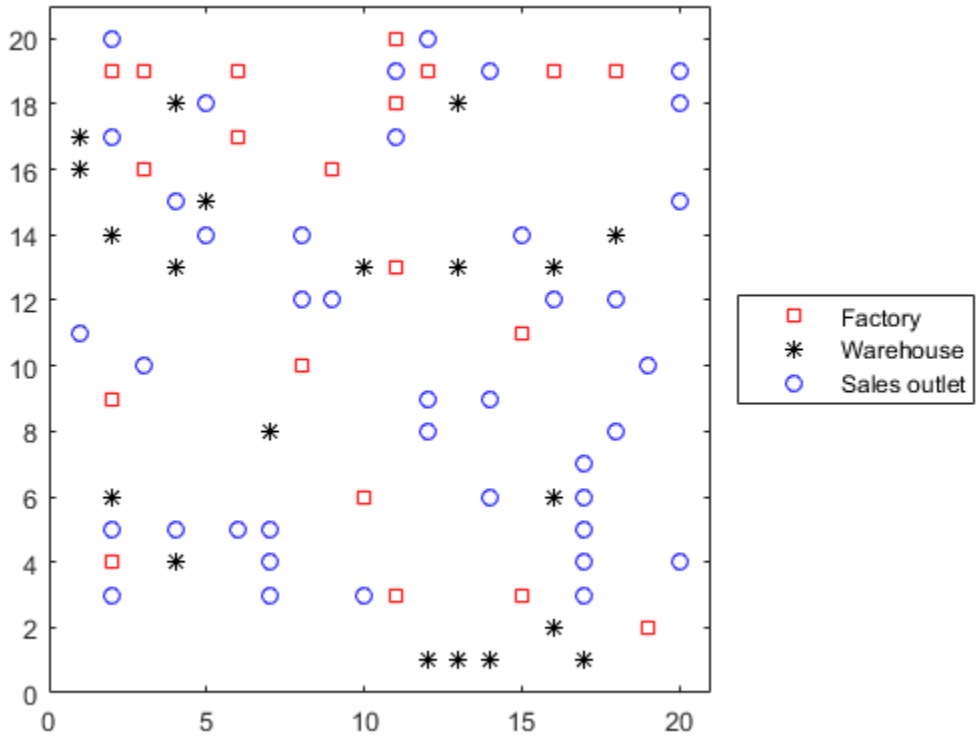

Generate Random Capacities, Costs, and Demands

Generate random production costs, capacities, turnover rates, and demands.

```
P = 20; % 20 products
```

```
% Production costs between 20 and 100
pcost = 80*rand(F,P) + 20;

% Production capacity between 500 and 1500 for each product/factory
pcap = 1000*rand(F,P) + 500;

% Warehouse capacity between P*400 and P*800 for each product/warehouse
wcap = P*400*rand(W,P) + P*400;

% Product turnover rate between 1 and 3 for each product
turn = 2*rand(1,P) + 1;

% Product transport cost per distance between 5 and 10 for each product
tcost = 5*rand(1,P) + 5;

% Product demand by sales outlet between 200 and 500 for each
% product/outlet
d = 300*rand(S,P) + 200;
```

These random demands and capacities can lead to infeasible problems. In other words, sometimes the demand exceeds the production and warehouse capacity constraints. If you alter some parameters and get an infeasible problem, during solution you will get an exitflag of -2.

Generate Objective and Constraint Matrices and Vectors

The objective function vector `obj` in `intlincon` consists of the coefficients of the variables $x(p, f, w)$ and $y(s, w)$. So there are naturally `P*F*W + S*W` coefficients in `obj`.

One way to generate the coefficients is to begin with a `P-by-F-by-W` array `obj1` for the x coefficients, and an `S-by-W` array `obj2` for the $y(s, w)$ coefficients. Then convert these arrays to two vectors and combine them into `obj` by calling

```
obj = [obj1(:);obj2(:)];

obj1 = zeros(P,F,W); % Allocate arrays
obj2 = zeros(S,W);
```

Throughout the generation of objective and constraint vectors and matrices, we generate the (p, f, w) array or the (s, w) array, and then convert the result to a vector.

To begin generating the inputs, generate the distance arrays `distfw(i,j)` and `distsw(i,j)`.

```matlab
distfw = zeros(F,W); % Allocate matrix for factory-warehouse distances
for ii = 1:F
    for jj = 1:W
        distfw(ii,jj) = abs(xloc(ii) - xloc(F + jj)) + abs(yloc(ii) ...
            - yloc(F + jj));
    end
end

distsw = zeros(S,W); % Allocate matrix for sales outlet-warehouse distances
for ii = 1:S
    for jj = 1:W
        distsw(ii,jj) = abs(xloc(F + W + ii) - xloc(F + jj)) ...
            + abs(yloc(F + W + ii) - yloc(F + jj));
    end
end
```

Generate the entries of `obj1` and `obj2`.

```matlab
for ii = 1:P
    for jj = 1:F
        for kk = 1:W
            obj1(ii,jj,kk) = pcost(jj,ii) + tcost(ii)*distfw(jj,kk);
        end
    end
end

for ii = 1:S
    for jj = 1:W
        obj2(ii,jj) = distsw(ii,jj)*sum(d(ii,:).*tcost);
    end
end
```

Combine the entries into one vector.

```matlab
obj = [obj1(:);obj2(:)]; % obj is the objective function vector
```

Now create the constraint matrices.

The width of each linear constraint matrix is the length of the `obj` vector.

```matlab
matwid = length(obj);
```

There are two types of linear inequalities: the production capacity constraints, and the warehouse capacity constraints.

There are `P*F` production capacity constraints, and `W` warehouse capacity constraints. The constraint matrices are quite sparse, on the order of 1% nonzero, so save memory by using sparse matrices.

```matlab
Aineq = spalloc(P*F + W,matwid,P*F*W + S*W); % Allocate sparse Aeq
bineq = zeros(P*F + W,1); % Allocate bineq as full

% Zero matrices of convenient sizes:
clearer1 = zeros(size(obj1));
clearer12 = clearer1(:);
clearer2 = zeros(size(obj2));
clearer22 = clearer2(:);

% First the production capacity constraints
counter = 1;
for ii = 1:F
    for jj = 1:P
        xtemp = clearer1;
        xtemp(jj,ii,:) = 1; % Sum over warehouses for each product and factory
        xtemp = sparse([xtemp(:);clearer22]); % Convert to sparse
        Aineq(counter,:) = xtemp'; % Fill in the row
        bineq(counter) = pcap(ii,jj);
        counter = counter + 1;
    end
end

% Now the warehouse capacity constraints
vj = zeros(S,1); % The multipliers
for jj = 1:S
    vj(jj) = sum(d(jj,:)./turn); % A sum of P elements
end

for ii = 1:W
    xtemp = clearer2;
    xtemp(:,ii) = vj;
    xtemp = sparse([clearer12;xtemp(:)]); % Convert to sparse
    Aineq(counter,:) = xtemp'; % Fill in the row
    bineq(counter) = wcap(ii);
    counter = counter + 1;
end
```

There are two types of linear equality constraints: the constraint that demand is met, and the constraint that each sales outlet corresponds to one warehouse.

```matlab
Aeq = spalloc(P*W + S,matwid,P*W*(F+S) + S*W); % Allocate as sparse
```

```
beq = zeros(P*W + S,1); % Allocate vectors as full

counter = 1;
% Demand is satisfied:
for ii = 1:P
    for jj = 1:W
        xtemp = clearer1;
        xtemp(ii,:,jj) = 1;
        xtemp2 = clearer2;
        xtemp2(:,jj) = -d(:,ii);
        xtemp = sparse([xtemp(:);xtemp2(:)]'); % Change to sparse row
        Aeq(counter,:) = xtemp; % Fill in row
        counter = counter + 1;
    end
end

% Only one warehouse for each sales outlet:
for ii = 1:S
    xtemp = clearer2;
    xtemp(ii,:) = 1;
    xtemp = sparse([clearer12;xtemp(:)]'); % Change to sparse row
    Aeq(counter,:) = xtemp; % Fill in row
    beq(counter) = 1;
    counter = counter + 1;
end
```

Bound Constraints and Integer Variables

The integer variables are those from `length(obj1) + 1` to the end.

```
intcon = P*F*W+1:length(obj);
```

The upper bounds are from `length(obj1) + 1` to the end also.

```
lb = zeros(length(obj),1);
ub = Inf(length(obj),1);
ub(P*F*W+1:end) = 1;
```

Turn off iterative display so that you don't get hundreds of lines of output. Include a plot function to monitor the solution progress.

```
opts = optimoptions('intlinprog','Display','off','PlotFcns',@optimplotmilp);
```

Solve the Problem

You generated all the solver inputs. Call the solver to find the solution.

```
[solution,fval,exitflag,output] = intlinprog(obj,intcon,...
                                   Aineq,bineq,Aeq,beq,lb,ub,opts);
if isempty(solution) % If the problem is infeasible or you stopped early with no soluti
    disp('intlinprog did not return a solution.')
    return % Stop the script because there is nothing to examine
end
```
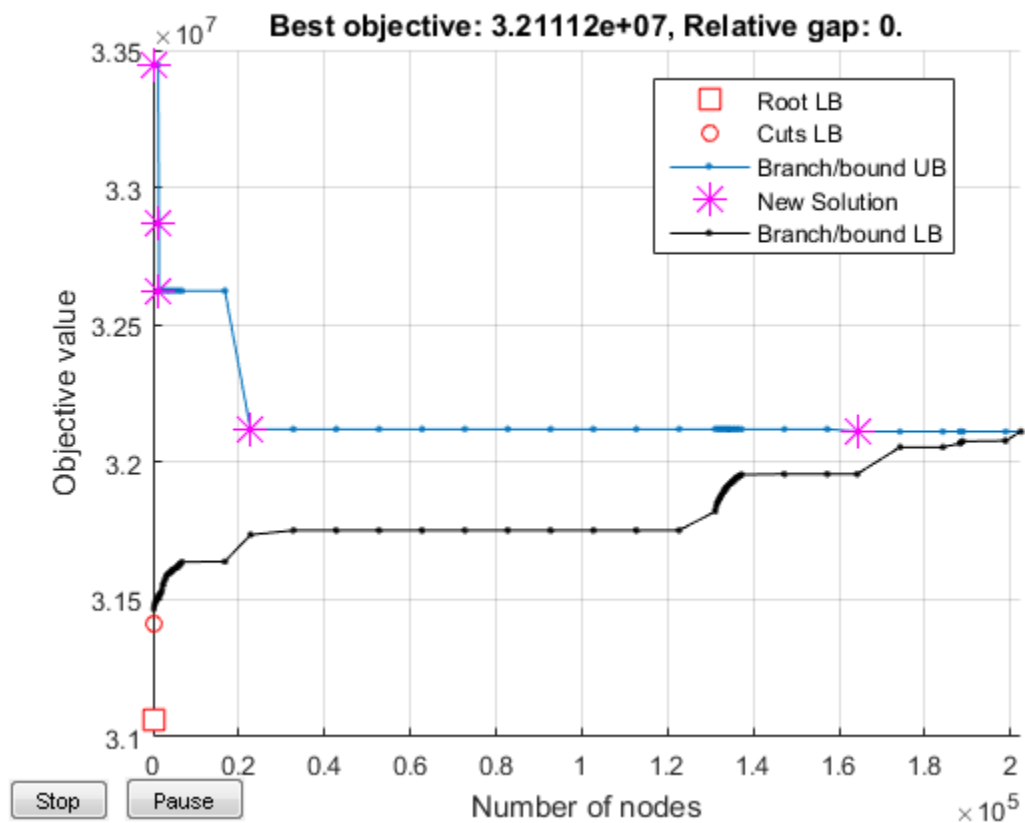

Examine the Solution

The solution is feasible, to within the given tolerances.

```
exitflag
infeas1 = max(Aineq*solution - bineq)
infeas2 = norm(Aeq*solution - beq,Inf)
```

```
exitflag =

     1


infeas1 =

   1.1369e-12


infeas2 =

   7.9581e-13
```

Check that the integer components are really integers, or are close enough that it is reasonable to round them. To understand why these variables might not be exactly integers, see the documentation.

```
diffint = norm(solution(intcon) - round(solution(intcon)),Inf)


diffint =

   4.4409e-16
```

Some integer variables are not exactly integers, but all are very close. So round the integer variables.

```
solution(intcon) = round(solution(intcon));
```

Check the feasibility of the rounded solution, and the change in objective function value.

```
infeas1 = max(Aineq*solution - bineq)
infeas2 = norm(Aeq*solution - beq,Inf)
diffrounding = norm(fval - obj(:)'*solution,Inf)


infeas1 =

   1.1369e-12
```

```
infeas2 =

   1.0232e-12


diffrounding =

   7.4506e-09
```

Rounding the solution did not appreciably change its feasibility.

You can examine the solution most easily by reshaping it back to its original dimensions.

```
solution1 = solution(1:P*F*W); % The continuous variables
solution2 = solution(intcon); % The integer variables
solution1 = reshape(solution1,P,F,W);
solution2 = reshape(solution2,S,W);
```

For example, how many sales outlets are associated with each warehouse? Notice that, in this case, some warehouses have 0 associated outlets, meaning the warehouses are not in use in the optimal solution.

```
outlets = sum(solution2,1) % Sum over the sales outlets


outlets =

  Columns 1 through 13

    2    0    2    3    3    2    2    3    2    3    2    1    2

  Columns 14 through 20

    1    4    0    3    3    2    0
```

Plot the connection between each sales outlet and its warehouse.

```
figure(h);
hold on
for ii = 1:S
    jj = find(solution2(ii,:)); % Index of warehouse associated with ii
    xsales = xloc(F+W+ii); ysales = yloc(F+W+ii);
    xwarehouse = xloc(F+jj); ywarehouse = yloc(F+jj);
```

```matlab
    if rand(1) < .5 % Draw y direction first half the time
        plot([xsales,xsales,xwarehouse],[ysales,ywarehouse,ywarehouse],'g--')
    else % Draw x direction first the rest of the time
        plot([xsales,xwarehouse,xwarehouse],[ysales,ysales,ywarehouse],'g--')
    end
end
hold off

title('Mapping of sales outlets to warehouses')
```


Mapping of sales outlets to warehouses

The black * with no green lines represent the unused warehouses.

Travelling Salesman Problem

This example shows how to use binary integer programming to solve the classic travelling salesman problem. This problem involves finding the shortest closed tour (path) through a set of stops (cities). In this case there are 200 stops, but you can easily change the nStops variable to get a different problem size. You'll solve the initial problem and see that the solution has subtours. This means the optimal solution found doesn't give one continuous path through all the points, but instead has several disconnected loops. You'll then use an iterative process of determining the subtours, adding constraints, and rerunning the optimization until the subtours are eliminated.

Draw the Map and Stops

Generate random stops inside a crude polygonal representation of the continental U.S.

```matlab
figure;

load('usborder.mat','x','y','xx','yy');
rng(3,'twister') % makes a plot with stops in Maine & Florida, and is reproducible
nStops = 200; % you can use any number, but the problem size scales as N^2
stopsLon = zeros(nStops,1); % allocate x-coordinates of nStops
stopsLat = stopsLon; % allocate y-coordinates
n = 1;
while (n <= nStops)
    xp = rand*1.5;
    yp = rand;
    if inpolygon(xp,yp,xx,yy) % test if inside the border
        stopsLon(n) = xp;
        stopsLat(n) = yp;
        n = n+1;
    end
end
plot(x,y,'Color','red'); % draw the outside border
hold on
% Add the stops to the map
plot(stopsLon,stopsLat,'*b')
hold off
```
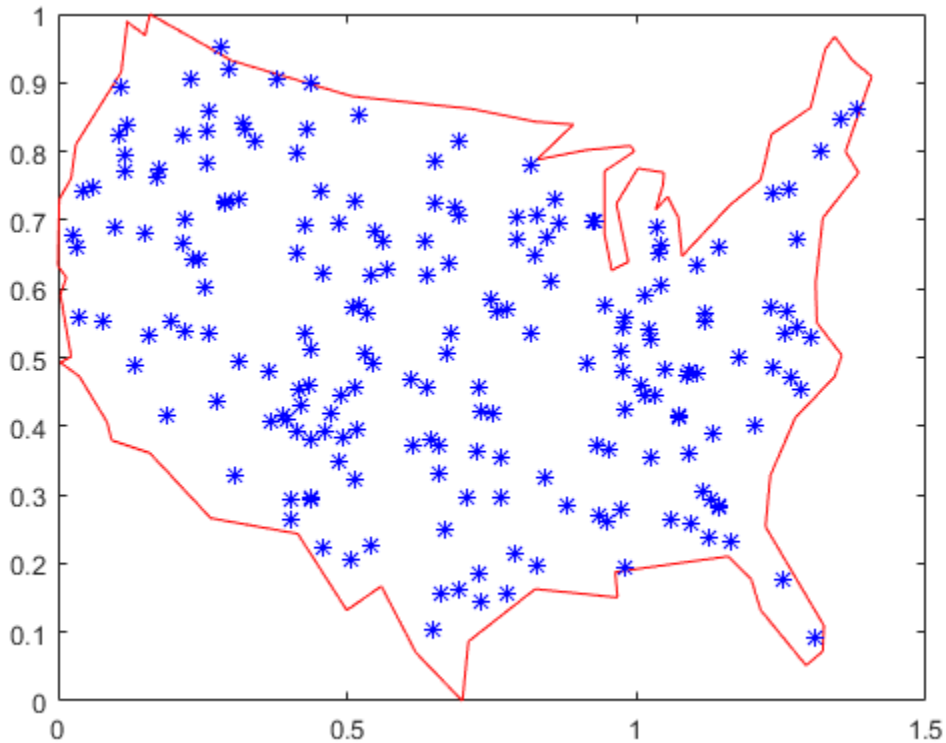
Problem Formulation

Formulate the travelling salesman problem for integer linear programming as follows:

- Generate all possible trips, meaning all distinct pairs of stops.

- Calculate the distance for each trip.

- The cost function to minimize is the sum of the trip distances for each trip in the tour.

- The decision variables are binary, and associated with each trip, where each 1 represents a trip that exists on the tour, and each 0 represents a trip that is not on the tour.

- To ensure that the tour includes every stop, include the linear constraint that each stop is on exactly two trips. This means one arrival and one departure from the stop.

Calculate Distances Between Points

Because there are 200 stops, there are 19,900 trips, meaning 19,900 binary variables (# variables = 200 choose 2).

Generate all the trips, meaning all pairs of stops.

```
idxs = nchoosek(1:nStops,2);
```

Calculate all the trip distances, assuming that the earth is flat in order to use the Pythagorean rule.

```
dist = hypot(stopsLat(idxs(:,1)) - stopsLat(idxs(:,2)), ...
             stopsLon(idxs(:,1)) - stopsLon(idxs(:,2)));
lendist = length(dist);
```

With this definition of the `dist` vector, the length of a tour is

```
dist'*x_tsp
```

where `x_tsp` is the binary solution vector. This is the distance of a tour that you try to minimize.

Equality Constraints

The problem has two types of equality constraints. The first enforces that there must be 200 trips total. The second enforces that each stop must have two trips attached to it (there must be a trip to each stop and a trip departing each stop).

Specify the first type of equality constraint, that you must have `nStops` trips, in the form `Aeq*x_tsp = beq`.

```
Aeq = spones(1:length(idxs)); % Adds up the number of trips
beq = nStops;
```

To specify the second type of equality constraint, that there needs to be two trips attached to each stop, extend the `Aeq` matrix as sparse.

```
Aeq = [Aeq;spalloc(nStops,length(idxs),nStops*(nStops-1))]; % allocate a sparse matrix
```

```
for ii = 1:nStops
    whichIdxs = (idxs == ii); % find the trips that include stop ii
    whichIdxs = sparse(sum(whichIdxs,2)); % include trips where ii is at either end
    Aeq(ii+1,:) = whichIdxs'; % include in the constraint matrix
end
beq = [beq; 2*ones(nStops,1)];
```

Binary Bounds

All decision variables are binary. Now, set the `intcon` argument to the number of decision variables, put a lower bound of 0 on each, and an upper bound of 1.

```
intcon = 1:lendist;
lb = zeros(lendist,1);
ub = ones(lendist,1);
```

Optimize Using intlinprog

The problem is ready to be solved. Call the solver.

```
opts = optimoptions('intlinprog','Display','off');
[x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,[],[],Aeq,beq,lb,ub,opts);
```

Visualize the Solution

```
hold on
segments = find(x_tsp); % Get indices of lines on optimal path
lh = zeros(nStops,1); % Use to store handles to lines on plot
lh = updateSalesmanPlot(lh,x_tsp,idxs,stopsLon,stopsLat);
title('Solution with Subtours');
```
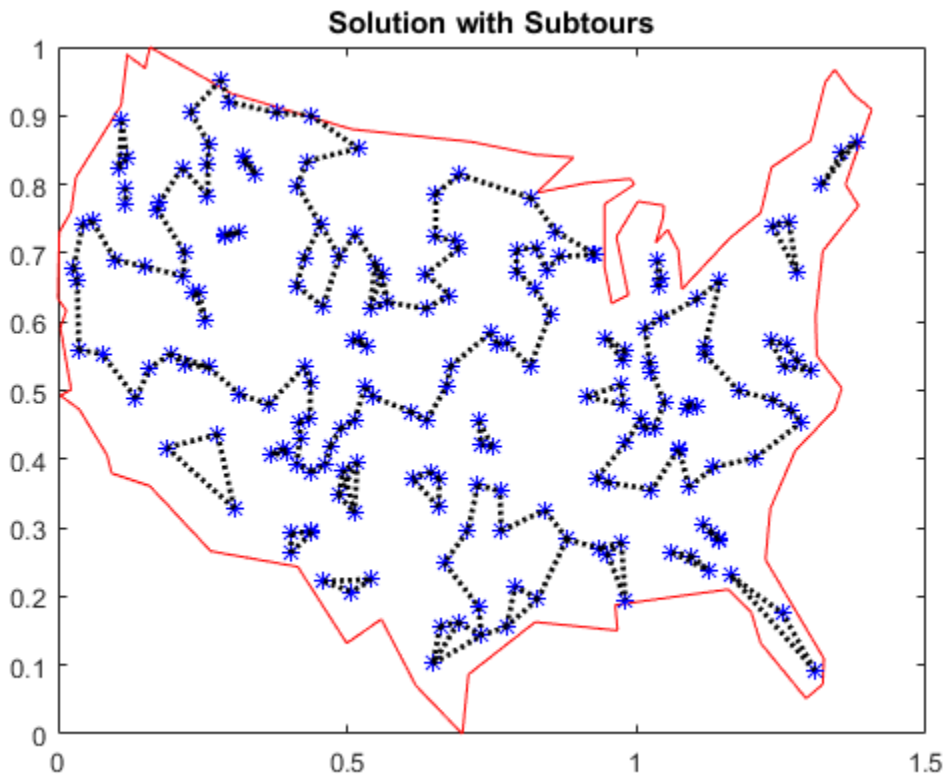
As can be seen on the map, the solution has several subtours. The constraints specified so far do not prevent these subtours from happening. In order to prevent any possible subtour from happening, you would need an incredibly large number of inequality constraints.

Subtour Constraints

Because you can't add all of the subtour constraints, take an iterative approach. Detect the subtours in the current solution, then add inequality constraints to prevent those particular subtours from happening. By doing this, you find a suitable tour in a few iterations.

Eliminate subtours with inequality constraints. An example of how this works is if you have five points in a subtour, then you have five lines connecting those points to create the subtour. Eliminate this subtour by implementing an inequality constraint to say there must be less than or equal to four lines between these five points.

Even more, find all lines between these five points, and constrain the solution not to have more than four of these lines present. This is a correct constraint because if five or more of the lines existed in a solution, then the solution would have a subtour (a graph with n nodes and n edges always contains a cycle).

The `detectSubtours` function analyzes the solution and returns a cell array of vectors. Each vector in the cell array contains the stops involved in that particular subtour.

```
tours = detectSubtours(x_tsp,idxs);
numtours = length(tours); % number of subtours
fprintf('# of subtours: %d\n',numtours);

# of subtours: 28
```

Include the linear inequality constraints to eliminate subtours, and repeatedly call the solver, until just one subtour remains.

```
A = spalloc(0,lendist,0); % Allocate a sparse linear inequality constraint matrix
b = [];
while numtours > 1 % repeat until there is just one subtour
    % Add the subtour constraints
    b = [b;zeros(numtours,1)]; % allocate b
    A = [A;spalloc(numtours,lendist,nStops)]; % a guess at how many nonzeros to allocat
    for ii = 1:numtours
        rowIdx = size(A,1)+1; % Counter for indexing
        subTourIdx = tours{ii}; % Extract the current subtour
%         The next lines find all of the variables associated with the
%         particular subtour, then add an inequality constraint to prohibit
%         that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx),2);
        for jj = 1:length(variations)
            whichVar = (sum(idxs==subTourIdx(variations(jj,1)),2)) & ...
                       (sum(idxs==subTourIdx(variations(jj,2)),2));
            A(rowIdx,whichVar) = 1;
        end
        b(rowIdx) = length(subTourIdx)-1; % One less trip than subtour stops
    end
```

```
    % Try to optimize again
    [x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,A,b,Aeq,beq,lb,ub,opts);

    % Visualize result
    lh = updateSalesmanPlot(lh,x_tsp,idxs,stopsLon,stopsLat);

    % How many subtours this time?
    tours = detectSubtours(x_tsp,idxs);
    numtours = length(tours); % number of subtours
    fprintf('# of subtours: %d\n',numtours);
end

title('Solution with Subtours Eliminated');
hold off

# of subtours: 22
# of subtours: 10
# of subtours: 5
# of subtours: 3
# of subtours: 4
# of subtours: 1
```

Solution with Subtours Eliminated
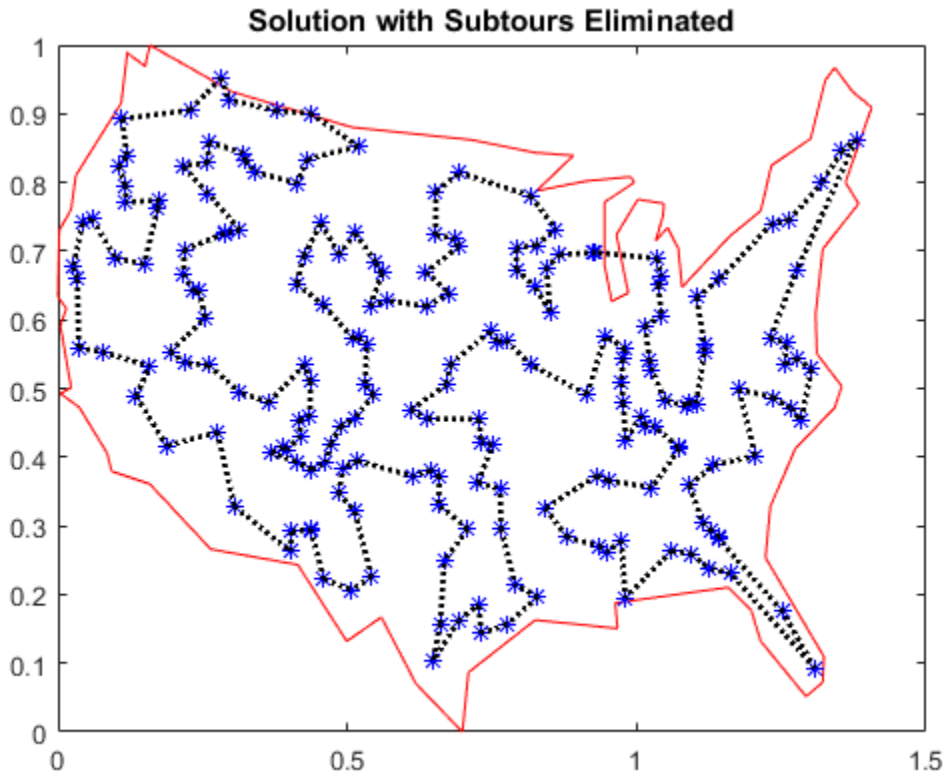
Solution Quality

The solution represents a feasible tour, because it is a single closed loop. But is it a minimal-cost tour? One way to find out is to examine the output structure.

```
disp(output.absolutegap)
```

    0.0021

The smallness of the absolute gap implies that the solution is either optimal or has a total length that is close to optimal.

Optimal Dispatch of Power Generators

This example shows how to schedule two gas-fired electric generators optimally, meaning to get the most revenue minus cost. While the example is not entirely realistic, it does show how to take into account costs that depend on decision timing.

Problem Definition

The electricity market has different prices at different times of day. If you have generators, you can take advantage of this variable pricing by scheduling your generators to operate when prices are high. Suppose that there are two generators that you control. Each generator has three power levels (off, low, and high). Each generator has a specified rate of fuel consumption and power production at each power level. Of course, fuel consumption is 0 when the generator is off.

You can assign a power level to each generator during each half-hour time interval during a day (24 hours, so 48 intervals). Based on historical records, you can assume that you know the revenue per megawatt-hour (MWh) that you get in each time interval. The data for this example is from the Australian Energy Market Operator `http://www.nemweb.com.au/REPORTS/CURRENT/` in mid-2013, and is used under their terms `http://www.aemo.com.au/About-AEMO/Legal-Notices/Copyright-Permissions`.

```
load dispatchPrice; % Get poolPrice, which is the revenue per MWh
bar(poolPrice,.5)
xlim([.5,48.5])
xlabel('Price per MWh at each period')
```

Price per MWh at each period
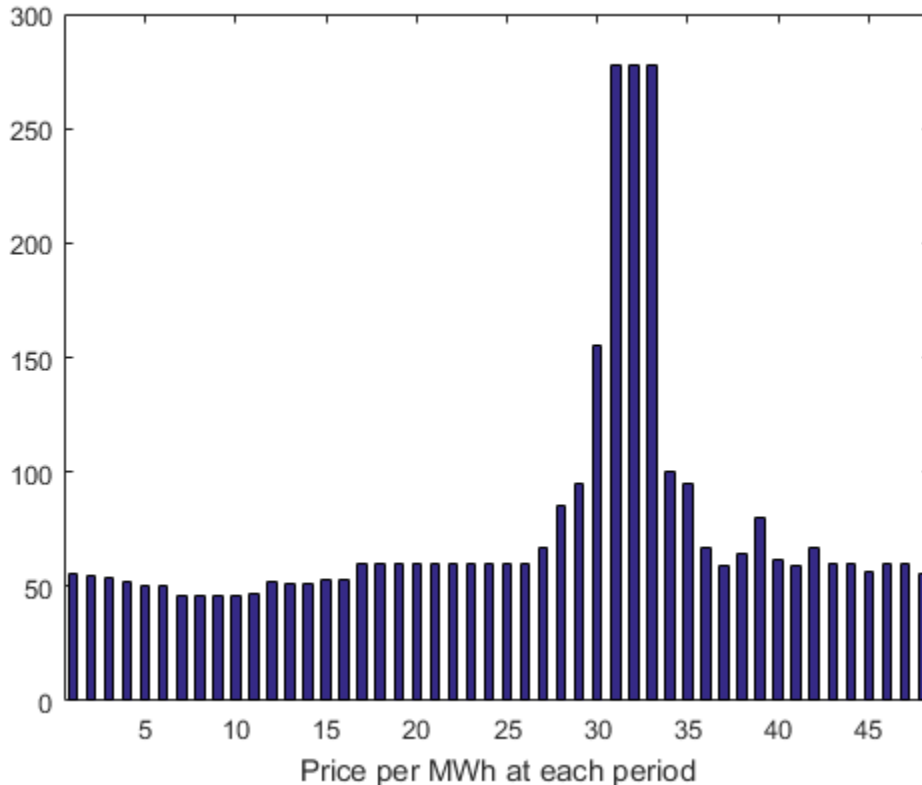
There is a cost to start a generator after it has been off. The other constraint is a maximum fuel usage for the day. The maximum fuel constraint is because you buy your fuel a day ahead of time, so can use only what you just bought.

Problem Notation and Parameters

You can formulate the scheduling problem as a binary integer programming problem as follows. Define indexes i, j, and k, and a binary scheduling vector y as:

- nPeriods = the number of time periods, 48 in this case.

- i = a time period, $1 \leq i \leq 48$.

- j = a generator index, $1 \leq j \leq 2$ for this example.

- `y(i,j,k) = 1` when period `i`, generator `j` is operating at power level `k`. Let low power be `k = 1`, and high power be `k = 2`. The generator is off when `sum_k y(i,j,k) = 0`.

You need to determine when a generator starts after being off. Let

- `z(i,j) = 1` when generator `j` is off at period `i`, but is on at period `i + 1`. `z(i,j) = 0` otherwise. In other words, `z(i,j) = 1` when `sum_k y(i,j,k) = 0` and `sum_k y(i+1,j,k) = 1`.

Obviously, you need a way to set `z` automatically based on the settings of `y`. A linear constraint below handles this setting.

You also need the parameters of the problem for costs, generation levels for each generator, consumption levels of the generators, and fuel available.

- `poolPrice(i)` -- Revenue in dollars per MWh in interval `i`.
- `gen(j,k)` -- MW generated by generator `j` at power level `k`.
- `fuel(j,k)` -- Fuel used by generator `j` at power level `k`.
- `totalfuel` -- Fuel available in one day.
- `startCost` -- Cost in dollars to start a generator after it has been off.
- `fuelPrice` -- Cost for a unit of fuel.

You got `poolPrice` when you executed `load dispatchPrice;`. Set the other parameters as follows.

```
fuelPrice = 3;
totalfuel = 3.95e4;
nPeriods = length(poolPrice); % 48 periods
nGens = 2; % Two generators
gen = [61,152;50,150]; % Generator 1 low = 61 MW, high = 152 MW
fuel = [427,806;325,765]; % Fuel consumption for generator 2 is low = 325, high = 765
startCost = 1e4; % Cost to start a generator after it has been off
```

Generator Efficiency

Examine the efficiency of the two generators at their two operating points.

```
efficiency = gen./fuel; % Calculate electricity per unit fuel use
rr = efficiency'; % for plotting
```

```
h = bar(rr);
h(1).FaceColor = 'g';
h(2).FaceColor = 'c';
legend(h,'Generator 1','Generator 2','Location','NorthEastOutside')
ax = gca;
ax.XTick = [1,2];
ax.XTickLabel = {'Low','High'};
ylim([.1,.2])
ylabel('Efficiency')
```
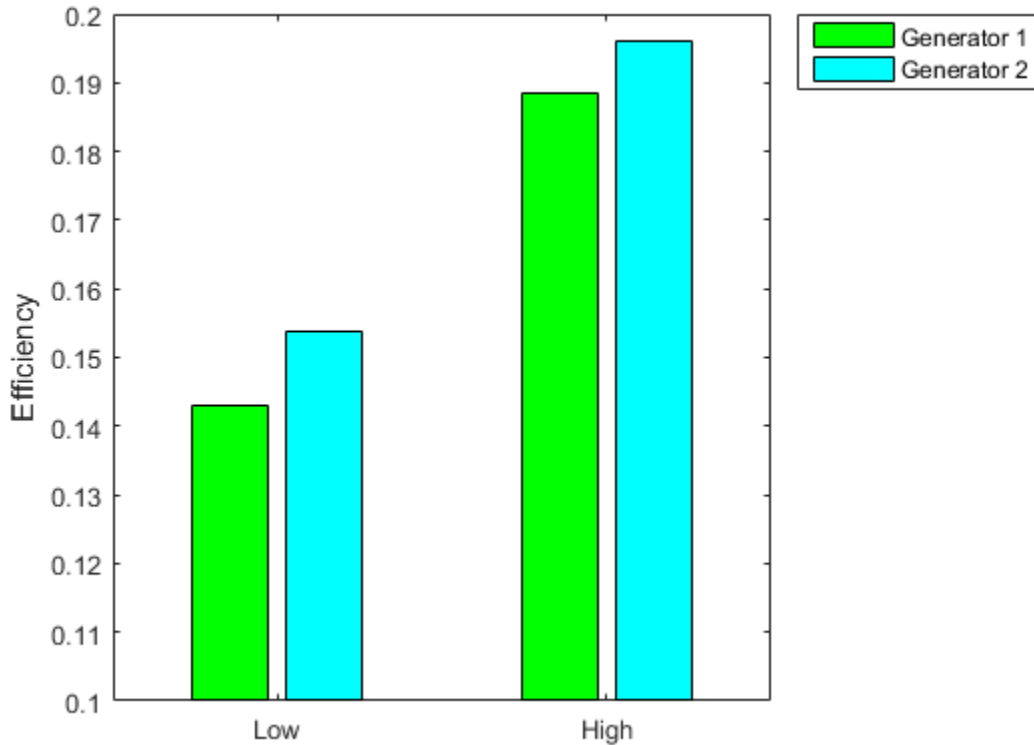

Notice that generator 2 is a bit more efficient than generator 1 at its corresponding operating points (low or high), but generator 1 at its high operating point is more efficient than generator 2 at its low operating point.

Variables for Solution

To set up the problem, you need to encode all the problem data and constraints in the form that the `intlinprog` solver requires. You have variables `y(i,j,k)` that represent the solution of the problem, and `z(i,j)` auxiliary variables for charging to turn on a generator. `y` is an `nPeriods-by-nGens-by-2` array, and `z` is an `nPeriods-by-nGens` array.

To put these variables in one long vector, define the variable of unknowns `x`:

```
x = [y(:);z(:)];
```

For bounds and linear constraints, it is easiest to use the natural array formulation of `y` and `z`, then convert the constraints to the total decision variable, the vector `x`.

Bounds

The solution vector `x` consists of binary variables. Set up the bounds `lb` and `ub`.

```
lby = zeros(nPeriods,nGens,2); % O for the y variables
lbz = zeros(nPeriods,nGens); % O for the z variables
lb = [lby(:);lbz(:)]; % Column vector lower bound
ub = ones(size(lb)); % Binary variables have lower bound O, upper bound 1
```

Linear Constraints

For linear constraints `A*x <= b`, the number of columns in the `A` matrix must be the same as the length of `x`, which is the same as the length of `lb`. To create rows of `A` of the appropriate size, create zero matrices of the sizes of the `y` and `z` matrices.

```
cleary = zeros(nPeriods,nGens,2);
clearz = zeros(nPeriods,nGens);
```

To ensure that the power level has no more than one component equal to 1, set a linear inequality constraint:

```
x(i,j,1) + x(i,j,2) <= 1

A = spalloc(nPeriods*nGens,length(lb),2*nPeriods*nGens); % nPeriods*nGens inequalities
counter = 1;
for ii = 1:nPeriods
    for jj = 1:nGens
```

```
            temp = cleary;
            temp(ii,jj,:) = 1;
            addrow = [temp(:);clearz(:)]';
            A(counter,:) = sparse(addrow);
            counter = counter + 1;
        end
end
b = ones(nPeriods*nGens,1); % A*x <= b means no more than one of x(i,j,1) and x(i,j,2)
```

The running cost per period is the cost for fuel for that period. For generator j operating at level k, the cost is `fuelPrice * fuel(j,k)`.

To ensure that the generators do not use too much fuel, create an inequality constraint on the sum of fuel usage.

```
yFuel = lby; % Initialize fuel usage array
yFuel(:,1,1) = fuel(1,1); % Fuel use of generator 1 in low setting
yFuel(:,1,2) = fuel(1,2); % Fuel use of generator 1 in high setting
yFuel(:,2,1) = fuel(2,1); % Fuel use of generator 2 in low setting
yFuel(:,2,2) = fuel(2,2); % Fuel use of generator 2 in high setting

addrow = [yFuel(:);clearz(:)]';
A = [A;sparse(addrow)];
b = [b;totalfuel]; % A*x <= b means the total fuel usage is <= totalfuel
```

Set the Generator Startup Indicator Variables

How can you get the solver to set the z variables automatically to match the active/off periods that the y variables represent? Recall that the condition to satisfy is `z(i,j) = 1` exactly when

`sum_k y(i,j,k) = 0` and `sum_k y(i+1,j,k) = 1`.

Notice that

`sum_k ( - y(i,j,k) + y(i+1,j,k) ) > 0` exactly when you want `z(i,j) = 1`.

Therefore, include the the linear inequality constraints

`sum_k ( - y(i,j,k) + y(i+1,j,k) ) - z(i,j) < = 0`

in the problem formulation, and include the z variables in the objective function cost. By including the z variables in the objective function, the solver attempts to lower the values

of the z variables, meaning it tries to set them all equal to 0. But for those intervals when a generator turns on, the linear inequality forces the z(i,j) to equal 1.

Add extra rows to the linear inequality constraint matrix A to represent these new inequalities. Wrap around the time so that interval 1 logically follows interval 48.

```
tempA = spalloc(nPeriods*nGens,length(lb),2*nPeriods*nGens);
counter = 1;
for ii = 1:nPeriods
    for jj = 1:nGens
        temp = cleary;
        tempy = clearz;
        temp(ii,jj,1) = -1;
        temp(ii,jj,2) = -1;
        if ii < nPeriods % Intervals 1 to 47
            temp(ii+1,jj,1) = 1;
            temp(ii+1,jj,2) = 1;
        else % Interval 1 follows interval 48
            temp(1,jj,1) = 1;
            temp(1,jj,2) = 1;
        end
        tempy(ii,jj) = -1;
        temp = [temp(:);tempy(:)]'; % Row vector for inclusion in tempA matrix
        tempA(counter,:) = sparse(temp);
        counter = counter + 1;
    end
end
A = [A;tempA];
b = [b;zeros(nPeriods*nGens,1)]; % A*x <= b sets z(i,j) = 1 at generator startup
```

Sparsity of Constraints

If you have a large problem, using sparse constraint matrices saves memory, and can save computational time as well. The constraint matrix A is quite sparse:

```
filledfraction = nnz(A)/numel(A)
```

```
filledfraction =

    0.0155
```

intlinprog accepts sparse linear constraint matrices A and Aeq, but requires their corresponding vector constraints b and beq to be full.

Define Objective

The objective function includes fuel costs for running the generators, revenue from running the generators, and costs for starting the generators.

```
generatorlevel = lby; % Generation in MW, start with Os
generatorlevel(:,1,1) = gen(1,1); % Fill in the levels
generatorlevel(:,1,2) = gen(1,2);
generatorlevel(:,2,1) = gen(2,1);
generatorlevel(:,2,2) = gen(2,2);
```

Incoming revenue = `x.*generatorlevel.*poolPrice`

```
revenue = generatorlevel; % Allocate revenue array
for ii = 1:nPeriods
    revenue(ii,:,:) = poolPrice(ii)*generatorlevel(ii,:,:);
end
```

Total fuel cost = `y.*yFuel*fuelPrice`

```
fuelCost = yFuel*fuelPrice;
```

Startup cost = `z.*ones(size(z))*startCost`

```
starts = (clearz + 1)*startCost;
starts = starts(:); % Generator startup cost vector
```

The vector `x = [y(:);z(:)]`. Write the total profit in terms of `x`:

profit = Incoming revenue - Total fuel cost - Startup cost

```
f = [revenue(:) - fuelCost(:);-starts]; % f is the objective function vector
```

Solve the Problem

To save space, suppress iterative display.

```
options = optimoptions('intlinprog','Display','final');
[x,fval,eflag,output] = intlinprog(-f,1:length(f),A,b,[],[],lb,ub,options);
```

```
Optimal solution found.
```

```
Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.TolGapAbs = 0 (the default value). The intcon variables
are integer within tolerance, options.TolInteger = 1e-05 (the default value).
```

Examine the Solution

The easiest way to examine the solution is dividing the solution vector x into its two components, y and z.

```
ysolution = x(1:nPeriods*nGens*2);
zsolution = x(nPeriods*nGens*2+1:end);
ysolution = reshape(ysolution,[nPeriods,nGens,2]);
zsolution = reshape(zsolution,[nPeriods,nGens]);
```

Plot the solution as a function of time.

```
subplot(3,1,1)
bar(ysolution(:,1,1)*gen(1,1)+ysolution(:,1,2)*gen(1,2),.5,'g')
xlim([.5,48.5])
ylabel('MWh')
title('Generator 1 optimal schedule','FontWeight','bold')
subplot(3,1,2)
bar(ysolution(:,2,1)*gen(1,1)+ysolution(:,2,2)*gen(1,2),.5,'c')
title('Generator 2 optimal schedule','FontWeight','bold')
xlim([.5,48.5])
ylabel('MWh')
subplot(3,1,3)
bar(poolPrice,.5)
xlim([.5,48.5])
title('Energy price','FontWeight','bold')
xlabel('Period')
ylabel('$ / MWh')
```
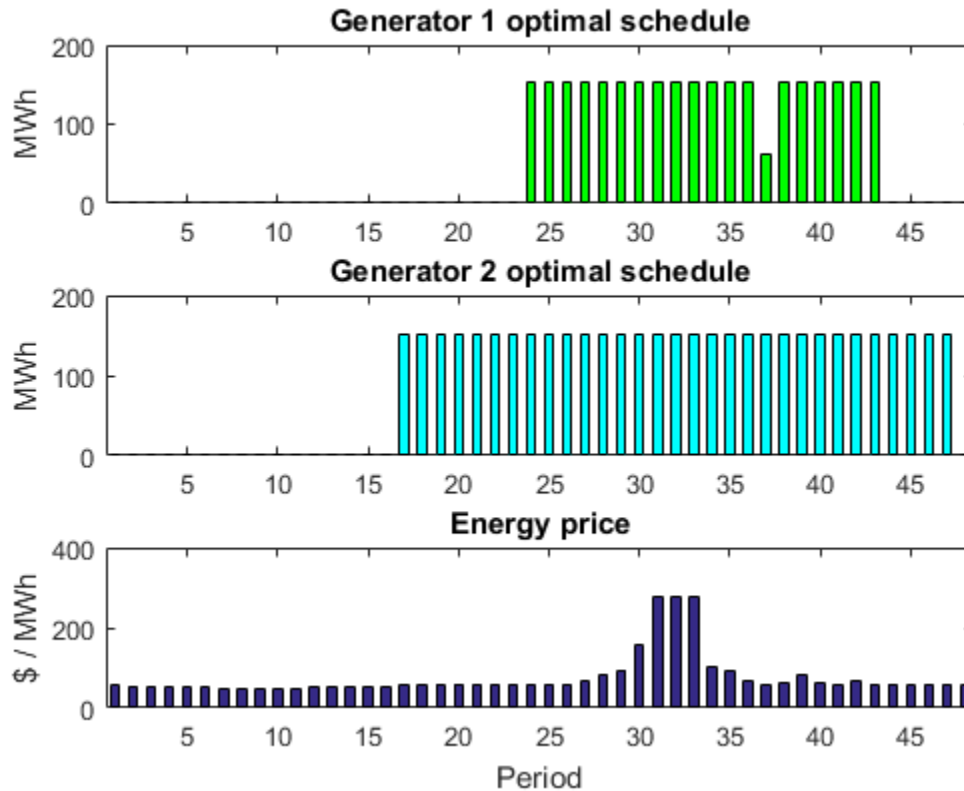
Generator 2 runs longer than generator 1, which you would expect because it is more efficient. Generator 2 runs at its high power level whenever it is on. Generator 1 runs mainly at its high power level, but dips down to low power for one time unit. Each generator runs for one contiguous set of periods daily, so incurs only one startup cost.

Check that the z variable is 1 for the periods when the generators start.

```
starttimes = find(round(zsolution) == 1); % Use round for noninteger results
[theperiod,thegenerator] = ind2sub(size(zsolution),starttimes)

theperiod =

    23
```

```
    16


thegenerator =

     1
     2
```

The periods when the generators start match the plots.

Compare to Lower Penalty for Startup

If you choose a small value of `startCost`, the solution involves multiple generation periods.

```
startCost = 500; % Choose a lower penalty for starting the generators
starts = (clearz + 1)*startCost;
starts = starts(:); % Start cost vector
fnew = [revenue(:) - fuelCost(:);-starts]; % New objective function
[xnew,fvalnew,eflagnew,outputnew] = ...
    intlinprog(-fnew,1:length(fnew),A,b,[],[],lb,ub,options);

ysolutionnew = xnew(1:nPeriods*nGens*2);
zsolutionnew = xnew(nPeriods*nGens*2+1:end);
ysolutionnew = reshape(ysolutionnew,[nPeriods,nGens,2]);
zsolutionnew = reshape(zsolutionnew,[nPeriods,nGens]);

subplot(3,1,1)
bar(ysolutionnew(:,1,1)*gen(1,1)+ysolutionnew(:,1,2)*gen(1,2),.5,'g')
xlim([.5,48.5])
ylabel('MWh')
title('Generator 1 optimal schedule','FontWeight','bold')
subplot(3,1,2)
bar(ysolutionnew(:,2,1)*gen(1,1)+ysolutionnew(:,2,2)*gen(1,2),.5,'c')
title('Generator 2 optimal schedule','FontWeight','bold')
xlim([.5,48.5])
ylabel('MWh')
subplot(3,1,3)
bar(poolPrice,.5)
xlim([.5,48.5])
title('Energy price','FontWeight','bold')
xlabel('Period')
ylabel('$ / MWh')
```

```matlab
starttimes = find(round(zsolutionnew) == 1); % Use round for noninteger results
[theperiod,thegenerator] = ind2sub(size(zsolution),starttimes)
```

```
Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.TolGapAbs = 0 (the default value). The intcon variables
are integer within tolerance, options.TolInteger = 1e-05 (the default value).


theperiod =

    22
    16
    45


thegenerator =

     1
     2
     2
```
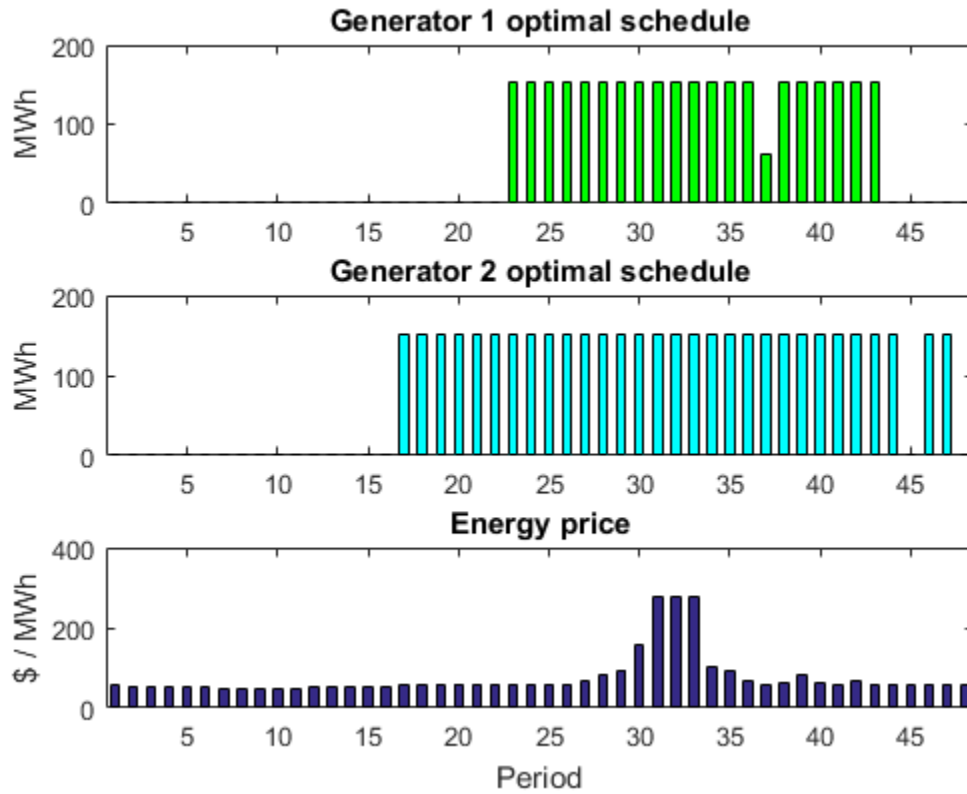
Mixed-Integer Quadratic Programming Portfolio Optimization

This example shows how to solve a Mixed-Integer Quadratic Programming (MIQP) portfolio optimization problem using the `intlinprog` Mixed-Integer Linear Programming (MILP) solver. The idea is to iteratively solve a sequence of MILP problems that locally approximate the MIQP problem.

Problem Outline

As Markowitz showed ("Portfolio Selection," J. Finance Volume 7, Issue 1, pp. 77-91, March 1952), you can express many portfolio optimization problems as quadratic programming problems. Suppose that you have a set of N assets and want to choose a portfolio, with $x(i)$ being the fraction of your investment that is in asset i. If you know the vector r of mean returns of each asset, and the covariance matrix Q of the returns, then for a given level of risk-aversion λ you maximize the risk-adjusted expected return:

$$\max_x (r^T x - \lambda x^T Q x).$$

The `quadprog` solver addresses this quadratic programming problem. However, in addition to the plain quadratic programming problem, you might want to restrict a portfolio in a variety of ways, such as:

- Having no more than M assets in the portfolio, where `M <= N`.

- Having at least m assets in the portfolio, where `0 < m <= M`.

- Having *semicontinuous* constraints, meaning either $x(i) = 0$, or $f\min \le x(i) \le f\max$ for some fixed fractions $f\min > 0$ and $f\max \ge f\min$.

You cannot include these constraints in `quadprog`. The difficulty is the discrete nature of the constraints. Furthermore, while the mixed-integer linear programming solver `intlinprog` does handle discrete constraints, it does not address quadratic objective functions.

This example constructs a sequence of MILP problems that satisfy the constraints, and that increasingly approximate the quadratic objective function. While this technique works for this example, it might not apply to different problem or constraint types.

Begin by modeling the constraints.

Modeling Discrete Constraints

x is the vector of asset allocation fractions, with $0 \leq x(i) \leq 1$ for each i. To model the number of assets in the portfolio, you need indicator variables v such that $v(i) = 0$ when $x(i) = 0$, and $v(i) = 1$ when $x(i) > 0$. To get variables that satisfy this restriction, set the v vector to be a binary variable, and impose the linear constraints

$$v(i)f\min \leq x(i) \leq v(i)f\max.$$

These inequalities both enforce that $x(i)$ and $v(i)$ are zero at exactly the same time, and they also enforce that $f\min \leq x(i) \leq f\max$ whenever $x(i) > 0$.

Also, to enforce the constraints on the number of assets in the portfolio, impose the linear constraints

$$m \leq \sum_i v(i) \leq M.$$

Objective and Successive Linear Approximations

As first formulated, you try to maximize the objective function. However, all Optimization Toolbox™ solvers minimize. So formulate the problem as minimizing the negative of the objective:

$$\min_x \lambda x^T Q x - r^T x.$$

This objective function is nonlinear. The `intlinprog` MILP solver requires a linear objective function. There is a standard technique to reformulate this problem into one with linear objective and nonlinear constraints. Introduce a slack variable z to represent the quadratic term.

$$\min_{x,z} \lambda z - r^T x \text{ such that } x^T Q x - z \leq 0, \ z \geq 0.$$

As you iteratively solve MILP approximations, you include new linear constraints, each of which approximates the nonlinear constraint locally near the current point. In particular, for $x = x_0 + \delta$ where x_0 is a constant vector and δ is a variable vector, the first-order Taylor approximation to the constraint is

$$x^T Q x - z = x_0^T Q x_0 + 2x_0^T Q \delta - z + O(|\delta|^2).$$

Replacing δ by $x - x_0$ gives

$$x^T Q x - z = -x_0^T Q x_0 + 2x_0^T Q x - z + O(|x - x_0|^2).$$

For each intermediate solution x_k you introduce a new linear constraint in x and z as the linear part of the expression above:

$$-x_k^T Q x_k + 2x_k^T Q x - z \le 0.$$

This has the form $Ax \le b$, where $A = 2x_k^T Q$, there is a -1 multiplier for the z term, and $b = x_k^T Q x_k$.

This method of adding new linear constraints to the problem is called a cutting plane method. For details, see J. E. Kelley, Jr. "The Cutting-Plane Method for Solving Convex Programs." J. Soc. Indust. Appl. Math. Vol. 8, No. 4, pp. 703-712, December, 1960.

MATLAB® Problem Formulation

To express problems for the `intlinprog` solver, you need to do the following:

- Decide what your variables represent
- Express lower and upper bounds in terms of these variables
- Give linear equality and inequality matrices

Have the first N variables represent the x vector, the next N variables represent the binary v vector, and the final variable represent the z slack variable. There are $2N + 1$ variables in the problem.

Load the data for the problem. This data has 225 expected returns in the vector `r` and the covariance of the returns in the 225-by-225 matrix `Q`. The data is the same as in the Using Quadratic Programming on Portfolio Optimization Problems example.

```
load port5
r = mean_return;
Q = Correlation .* (stdDev_return * stdDev_return');
```

Set the number of assets as `N`.

```
N = length(r);
```

Set indexes for the variables

```
xvars = 1:N;
vvars = N+1:2*N;
zvar = 2*N+1;
```

The lower bounds of all the 2N+1 variables in the problem are zero. The upper bounds of the first 2N variables are one, and the last variable has no upper bound.

```
lb = zeros(2*N+1,1);
ub = ones(2*N+1,1);
ub(zvar) = Inf;
```

Set the number of assets in the solution to be between 100 and 150. Incorporate this constraint into the problem in the form, namely

$$m \leq \sum_i v(i) \leq M,$$

by writing two linear constraints of the form $Ax \leq b$:

$$\sum_i v(i) \leq M$$

$$\sum_i -v(i) \leq -m.$$

```
M = 150;
m = 100;
A = zeros(1,2*N+1); % Allocate A matrix
A(vvars) = 1; % A*x represents the sum of the v(i)
A = [A;-A];
b = zeros(2,1); % Allocate b vector
b(1) = M;
b(2) = -m;
```

Include semicontinuous constraints. Take the minimal nonzero fraction of assets to be 0.001 for each asset type, and the maximal fraction to be 0.05.

```
fmin = 0.001;
```

```
fmax = 0.05;
```

Include the inequalities $x(i) \leq f\max(i) * v(i)$ and $f\min(i) * v(i) \leq x(i)$ as linear inequalities.

```
Atemp = eye(N);
Amax = horzcat(Atemp,-Atemp*fmax,zeros(N,1));
A = [A;Amax];
b = [b;zeros(N,1)];
Amin = horzcat(-Atemp,Atemp*fmin,zeros(N,1));
A = [A;Amin];
b = [b;zeros(N,1)];
```

Include the constraint that the portfolio is 100% invested, meaning $\sum x_i = 1$.

```
Aeq = zeros(1,2*N+1); % Allocate Aeq matrix
Aeq(xvars) = 1;
beq = 1;
```

Set the risk-aversion coefficient λ to 100.

```
lambda = 100;
```

Define the objective function $\lambda z - r^T x$ as a vector. Include zeros for the multipliers of the v variables.

```
f = [-r;zeros(N,1);lambda];
```

Solve the Problem

To solve the problem iteratively, begin by solving the problem with the current constraints, which do not yet reflect any linearization. The integer constraints are in the vvars vector.

```
options = optimoptions(@intlinprog,'Display','off'); % Suppress iterative display
[xLinInt,fval,exitFlagInt,output] = intlinprog(f,vvars,A,b,Aeq,beq,lb,ub,options);
```

Prepare a stopping condition for the iterations: stop when the slack variable z is within 0.01% of the true quadratic value.

```
thediff = 1e-4;
iter = 1; % iteration counter
assets = xLinInt(xvars); % the x variables
truequadratic = assets'*Q*assets;
```

```
zslack = xLinInt(zvar); % slack variable value
```

Keep a history of the computed true quadratic and slack variables for plotting.

```
history = [truequadratic,zslack];
```

Compute the quadratic and slack values. If they differ, then add another linear constraint and solve again.

In toolbox syntax, each new linear constraint $Ax \leq b$ comes from the linear approximation

$$-x_k^T Q x_k + 2x_k^T Q x - z \leq 0.$$

You see that the new row of $A = 2x_k^T Q$ and the new element in $b = x_k^T Q x_k$, with the z term represented by a -1 coefficient in A.

After you find a new solution, use a linear constraint halfway between the old and new solutions. This heuristic way of including linear constraints can be faster than simply taking the new solution. To use the solution instead of the halfway heuristic, comment the "Midway" line below, and uncomment the following one.

```
while abs((zslack - truequadratic)/truequadratic) > thediff % relative error
    newArow = horzcat(2*assets'*Q,zeros(1,N),-1); % Linearized constraint
    A = [A;newArow];
    b = [b;truequadratic];
    % Solve the problem with the new constraints
    [xLinInt,fval,exitFlagInt,output] = intlinprog(f,vvars,A,b,Aeq,beq,lb,ub,options);
    assets = (assets+xLinInt(xvars))/2; % Midway from the previous to the current
%     assets = xLinInt(xvars); % Use the previous line or this one
    truequadratic = assets'*Q*assets;
    zslack = xLinInt(zvar);
    history = [history;truequadratic,zslack];
    iter = iter + 1;
end
```

Examine the Solution and Convergence Rate

Plot the history of the slack variable and the quadratic part of the objective function to see how they converged.

```
plot(history)
```

```matlab
legend('Quadratic','Slack')
xlabel('Iteration number')
title('Quadratic and linear approximation (slack)')
```
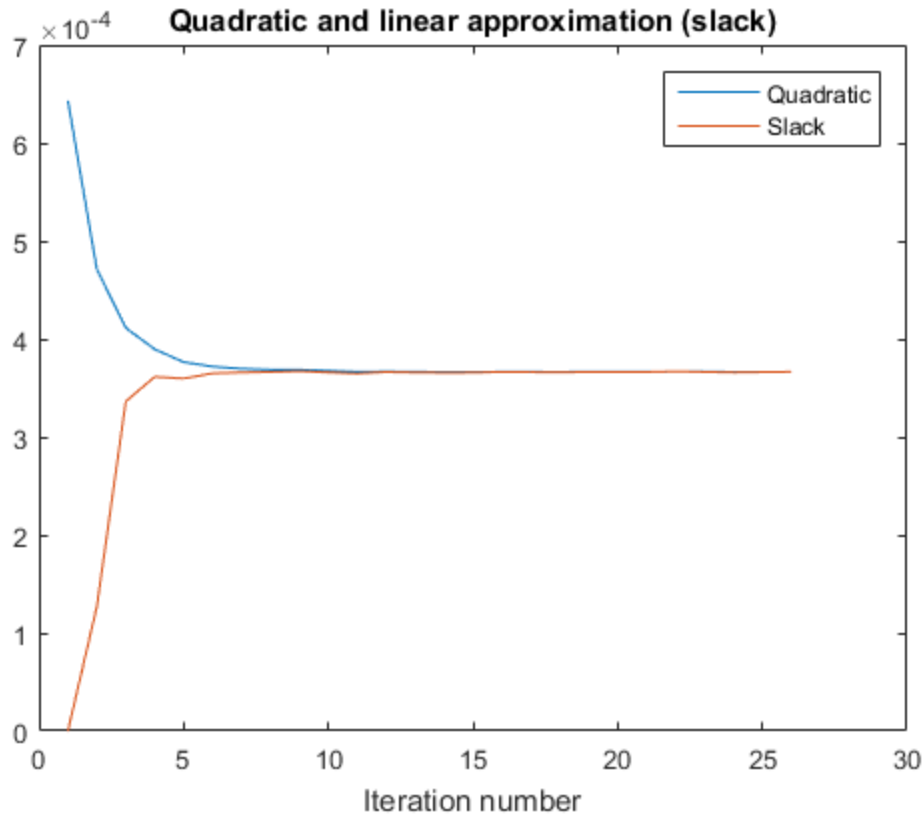

What is the quality of the MILP solution? The `output` structure contains that information. Examine the absolute gap between the internally-calculated bounds on the objective at the solution.

```matlab
disp(output.absolutegap)
```

```
0
```

The absolute gap is zero, indicating that the MILP solution is accurate.

Plot the optimal allocation. Use `xLinInt(xvars)`, not `assets`, because `assets` might not satisfy the constraints when using the midway update.

```
bar(xLinInt(xvars))
grid on
xlabel('Asset index')
ylabel('Proportion of investment')
title('Optimal asset allocation')
```
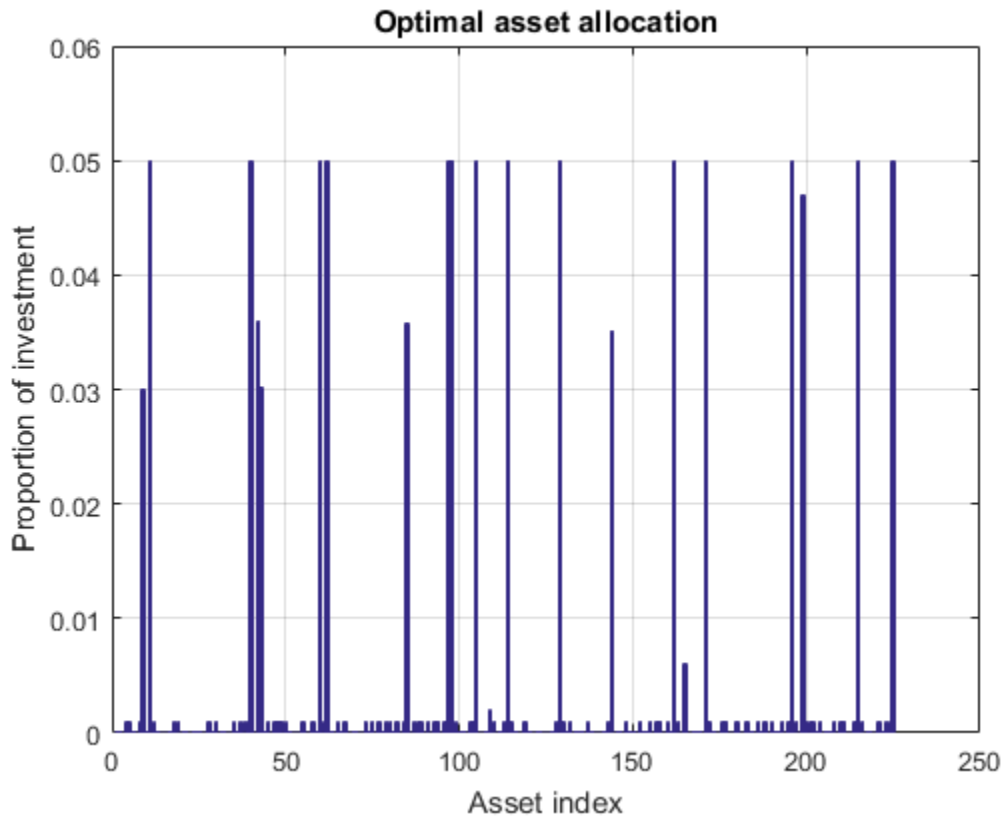

You can easily see that all nonzero asset allocations are between the semicontinuous bounds $f\text{min} = 0.001$ and $f\text{max} = 0.05$.

How many nonzero assets are there? The constraint is that there are between 100 and 150 nonzero assets.

```
sum(xLinInt(vvars))

ans =

   100
```

What is the expected return for this allocation, and the value of the risk-adjusted return?

```
fprintf('The expected return is %g, and the risk-adjusted return is %g.\n',...
    r'*xLinInt(xvars),-fval)
```

```
The expected return is 0.000616464, and the risk-adjusted return is -0.0360334.
```

More elaborate analyses are possible by using features specifically designed for portfolio optimization in Financial Toolbox™.

Solve Sudoku Puzzles Via Integer Programming

This example shows how to solve a Sudoku puzzle using binary integer programming.

You probably have seen Sudoku puzzles. A puzzle is to fill a 9-by-9 grid with integers from 1 through 9 so that each integer appears only once in each row, column, and major 3-by-3 square. The grid is partially populated with clues, and your task is to fill in the rest of the grid.

Initial Puzzle

Here is a data matrix B of clues. The first row, B(1,2,2), means row 1, column 2 has a clue 2. The second row, B(1,5,3), means row 1, column 5 has a clue 3. Here is the entire matrix B.

```
B = [1,2,2;
     1,5,3;
     1,8,4;
     2,1,6;
     2,9,3;
     3,3,4;
     3,7,5;
     4,4,8;
     4,6,6;
     5,1,8;
     5,5,1;
     5,9,6;
     6,4,7;
     6,6,5;
     7,3,7;
     7,7,6;
     8,1,4;
     8,9,8;
     9,2,3;
     9,5,4;
     9,8,2];

drawSudoku(B) % For the listing of this program, see the end of this example.
```

This puzzle, and an alternative MATLAB® solution technique, was featured in Cleve's Corner in 2009.

There are many approaches to solving Sudoku puzzles manually, as well as many programmatic approaches. This example shows a straightforward approach using binary integer programming.

This approach is particularly simple because you do not give a solution algorithm. Just express the rules of Sudoku, express the clues as constraints on the solution, and then `intlinprog` produces the solution.

Binary Integer Programming Approach

The key idea is to transform a puzzle from a square 9-by-9 grid to a cubic 9-by-9-by-9 array of binary values (0 or 1). Think of the cubic array as being 9 square grids stacked on top of each other. The top grid, a square layer of the array, has a 1 wherever the solution or clue has a 1. The second layer has a 1 wherever the solution or clue has a 2. The ninth layer has a 1 wherever the solution or clue has a 9.

This formulation is precisely suited for binary integer programming.

The objective function is not needed here, and might as well be 0. The problem is really just to find a feasible solution, meaning one that satisfies all the constraints. However, for tiebreaking in the internals of the integer programming solver, giving increased solution speed, use a nonconstant objective function.

Express the Rules for Sudoku as Constraints

Suppose a solution x is represented in a 9-by-9-by-9 binary array. What properties does x have? First, each square in the 2-D grid (i,j) has exactly one value, so there is exactly one nonzero element among the 3-D array entries $x(i,j,1),...,x(i,j,9)$. In other words, for every i and j,

$$\sum_{k=1}^{9} x(i,j,k) = 1.$$

Similarly, in each row i of the 2-D grid, there is exactly one value out of each of the digits from 1 to 9. In other words, for each i and k,

$$\sum_{j=1}^{9} x(i,j,k) = 1.$$

And each column j in the 2-D grid has the same property: for each j and k,

$$\sum_{i=1}^{9} x(i,j,k) = 1.$$

The major 3-by-3 grids have a similar constraint. For the grid elements $1 \leq i \leq 3$ and $1 \leq j \leq 3$, and for each $1 \leq k \leq 9$,

$$\sum_{i=1}^{3} \sum_{j=1}^{3} x(i,j,k) = 1.$$

To represent all nine major grids, just add 3 or 6 to each i and j index:

$$\sum_{i=1}^{3} \sum_{j=1}^{3} x(i+U, j+V, k) = 1,$$ where $U, V \epsilon \{0, 3, 6\}.$

Express Clues

Each initial value (clue) can be expressed as a constraint. Suppose that the (i, j) clue is m for some $1 \leq m \leq 9$. Then $x(i, j, m) = 1$. The constraint $\sum_{k=1}^{9} x(i, j, k) = 1$ ensures that all other $x(i, j, k) = 0$ for $k \neq m$.

Write the Rules for Sudoku

Although the Sudoku rules are conveniently expressed in terms of a 9-by-9-by-9 solution array x, linear constraints are given in terms of a vector solution matrix x(:). Therefore, when you write a Sudoku program, you have to use constraint matrices derived from 9-by-9-by-9 initial arrays.

Here is one approach to set up Sudoku rules, and also include the clues as constraints. The sudokuEngine file comes with your software.

type sudokuEngine

```
function [S,eflag] = sudokuEngine(B)
% This function sets up the rules for Sudoku. It reads in the puzzle
% expressed in matrix B, calls intlinprog to solve the puzzle, and returns
% the solution in matrix S.
%
% The matrix B should have 3 columns and at least 17 rows (because a Sudoku
% puzzle needs at least 17 entries to be uniquely solvable). The first two
% elements in each row are the i,j coordinates of a clue, and the third
% element is the value of the clue, an integer from 1 to 9. If B is a
% 9-by-9 matrix, the function first converts it to 3-column form.

%   Copyright 2014 The MathWorks, Inc.

if isequal(size(B),[9,9]) % 9-by-9 clues
    % Convert to 81-by-3
    [SM,SN] = meshgrid(1:9); % make i,j entries
    B = [SN(:),SM(:),B(:)]; % i,j,k rows
    % Now delete zero rows
    [rrem,~] = find(B(:,3) == 0);
    B(rrem,:) = [];
end

if size(B,2) ~= 3 || length(size(B)) > 2
    error('The input matrix must be N-by-3 or 9-by-9')
```

```
        end

        if sum([any(B ~= round(B)),any(B < 1),any(B > 9)]) % enforces entries 1-9
            error('Entries must be integers from 1 to 9')
        end

        %% The rules of Sudoku:
        N = 9^3; % number of independent variables in x, a 9-by-9-by-9 array
        M = 4*9^2; % number of constraints, see the construction of Aeq
        Aeq = zeros(M,N); % allocate equality constraint matrix Aeq*x = beq
        beq = ones(M,1); % allocate constant vector beq
        f = (1:N)'; % the objective can be anything, but having nonconstant f can speed the sol
        lb = zeros(9,9,9); % an initial zero array
        ub = lb+1; % upper bound array to give binary variables

        counter = 1;
        for j = 1:9 % one in each row
            for k = 1:9
                Astuff = lb; % clear Astuff
                Astuff(1:end,j,k) = 1; % one row in Aeq*x = beq
                Aeq(counter,:) = Astuff(:)'; % put Astuff in a row of Aeq
                counter = counter + 1;
            end
        end

        for i = 1:9 % one in each column
            for k = 1:9
                Astuff = lb;
                Astuff(i,1:end,k) = 1;
                Aeq(counter,:) = Astuff(:)';
                counter = counter + 1;
            end
        end

        for U = 0:3:6 % one in each square
            for V = 0:3:6
                for k = 1:9
                    Astuff = lb;
                    Astuff(U+(1:3),V+(1:3),k) = 1;
                    Aeq(counter,:) = Astuff(:)';
                    counter = counter + 1;
                end
            end
        end
```

```
for i = 1:9 % one in each depth
    for j = 1:9
        Astuff = lb;
        Astuff(i,j,1:end) = 1;
        Aeq(counter,:) = Astuff(:)';
        counter = counter + 1;
    end
end

%% Put the particular puzzle in the constraints
% Include the initial clues in the |lb| array by setting corresponding
% entries to 1. This forces the solution to have |x(i,j,k) = 1|.

for i = 1:size(B,1)
    lb(B(i,1),B(i,2),B(i,3)) = 1;
end

%% Solve the Puzzle
% The Sudoku problem is complete: the rules are represented in the |Aeq|
% and |beq| matrices, and the clues are ones in the |lb| array. Solve the
% problem by calling |intlinprog|. Ensure that the integer program has all
% binary variables by setting the intcon argument to |1:N|, with lower and
% upper bounds of 0 and 1.

intcon = 1:N;

[x,~,eflag] = intlinprog(f,intcon,[],[],Aeq,beq,lb,ub);

%% Convert the Solution to a Usable Form
% To go from the solution x to a Sudoku grid, simply add up the numbers at
% each $(i,j)$ entry, multiplied by the depth at which the numbers appear:

if eflag > 0 % good solution
    x = reshape(x,9,9,9); % change back to a 9-by-9-by-9 array
    x = round(x); % clean up non-integer solutions
    y = ones(size(x));
    for k = 2:9
        y(:,:,k) = k; % multiplier for each depth k
    end

    S = x.*y; % multiply each entry by its depth
    S = sum(S,3); % S is 9-by-9 and holds the solved puzzle
else
```

```
    S = [];
end
```

Call the Sudoku Solver

```
S = sudokuEngine(B); % Solves the puzzle pictured at the start
drawSudoku(S)
```

```
LP:              Optimal objective value is 29565.000000.

Cut Generation:  Applied 1 strong CG cut,
                 and 2 zero-half cuts.
                 Lower bound is 29565.000000.
                 Relative gap is 0.00%.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.TolGapAbs = 0 (the default value). The
intcon variables are integer within tolerance, options.TolInteger = 1e-05 (the
default value).
```

9	2	5	6	3	1	8	4	7
6	1	8	5	7	4	2	9	3
3	7	4	9	8	2	5	6	1
7	4	9	8	2	6	1	3	5
8	5	2	4	1	3	9	7	6
1	6	3	7	9	5	4	8	2
2	8	7	3	5	9	6	1	4
4	9	1	2	6	7	3	5	8
5	3	6	1	4	8	7	2	9

You can easily check that the solution is correct.

Function to Draw the Sudoku Puzzle

type drawSudoku

```
function drawSudoku(B)
% Function for drawing the Sudoku board

%   Copyright 2014 The MathWorks, Inc.


figure;hold on;axis off;axis equal % prepare to draw
```

```
rectangle('Position',[0 0 9 9],'LineWidth',3,'Clipping','off') % outside border
rectangle('Position',[3,0,3,9],'LineWidth',2) % heavy vertical lines
rectangle('Position',[0,3,9,3],'LineWidth',2) % heavy horizontal lines
rectangle('Position',[0,1,9,1],'LineWidth',1) % minor horizontal lines
rectangle('Position',[0,4,9,1],'LineWidth',1)
rectangle('Position',[0,7,9,1],'LineWidth',1)
rectangle('Position',[1,0,1,9],'LineWidth',1) % minor vertical lines
rectangle('Position',[4,0,1,9],'LineWidth',1)
rectangle('Position',[7,0,1,9],'LineWidth',1)

% Fill in the clues
%
% The rows of B are of the form (i,j,k) where i is the row counting from
% the top, j is the column, and k is the clue. To place the entries in the
% boxes, j is the horizontal distance, 10-i is the vertical distance, and
% we subtract 0.5 to center the clue in the box.
%
% If B is a 9-by-9 matrix, convert it to 3 columns first

if size(B,2) == 9 % 9 columns
    [SM,SN] = meshgrid(1:9); % make i,j entries
    B = [SN(:),SM(:),B(:)]; % i,j,k rows
end

for ii = 1:size(B,1)
    text(B(ii,2)-0.5,9.5-B(ii,1),num2str(B(ii,3)))
end

hold off

end
```

9

Quadratic Programming

Quadratic Programming Algorithms

Quadratic Programming 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 $A\,x \le b$, $Aeq\,x = beq$, $l \le x \le u$.

`interior-point-convex quadprog` Algorithm

The `interior-point-convex` algorithm performs the following steps:

1. "Presolve/Postsolve" on page 9-2
2. "Generate Initial Point" on page 9-3
3. "Predictor-Corrector" on page 9-3
4. "Multiple Corrections" on page 9-5
5. "Total Relative Error" on page 9-6

Presolve/Postsolve

The algorithm begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

- Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.

- Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.
- Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.
- Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.
- Check if the bounds and linear constraints are consistent.
- Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For details, see Gould and Toint [63].

Generate Initial Point

The initial point x0 for the algorithm is:

1 Initialize x0 to `ones(n,1)`, where n is the number of rows in *H*.
2 For components that have both an upper bound ub and a lower bound lb, if a component of x0 is not strictly inside the bounds, the component is set to `(ub + lb)/2`.
3 For components that have only one bound, modify the component if necessary to lie strictly inside the bound.

Predictor-Corrector

Similar to the `fmincon` interior-point algorithm, the `interior-point-convex` algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) conditions

hold. For the quadratic programming problem described in "Quadratic Programming Definition" on page 9-2, these conditions are:

$$Hx + c - A_{eq}^T y - \bar{A}^T z = 0$$
$$\bar{A}x - \bar{b} - s = 0$$
$$A_{eq}x - b_{eq} = 0$$
$$s_i z_i = 0, \ i = 1,2,...,m$$
$$s \geq 0$$
$$z \geq 0.$$

Here

- $\bar{A}$ is the extended linear inequality matrix that includes bounds written as linear inequalities. $\bar{b}$ is the corresponding linear inequality vector, including bounds.

- s is the vector of slacks that convert inequality constraints to equalities. s has length m, the number of linear inequalities and bounds.

- z is the vector of Lagrange multipliers corresponding to s.

- y is the vector of Lagrange multipliers associated with the equality constraints.

The algorithm first predicts a step from the Newton-Raphson formula, then computes a corrector step. The corrector attempts to better enforce the nonlinear constraint $s_i z_i = 0$.

Definitions for the predictor step:

- r_d, the dual residual:

$$r_d = Hx + c - A_{eq}^T y - \bar{A}^T z.$$

- r_{eq}, the primal equality constraint residual:

$$r_{eq} = A_{eq}x - b_{eq}.$$

- r_{ineq}, the primal inequality constraint residual, which includes bounds and slacks:

$$r_{ineq} = \bar{A}x - \bar{b} - s.$$

- r_{sz}, the complementarity residual:

  $r_{sz} = Sz$.

  S is the diagonal matrix of slack terms, z is the column matrix of Lagrange multipliers.

- r_c, the average complementarity:

$$r_c = \frac{s^T z}{m}.$$

In a Newton step, the changes in x, s, y, and z, are given by:

$$\begin{pmatrix} H & 0 & -A_{eq}^T & -\bar{A}^T \\ A_{eq} & 0 & 0 & 0 \\ \bar{A} & -I & 0 & 0 \\ 0 & Z & 0 & S \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta s \\ \Delta y \\ \Delta z \end{pmatrix} = - \begin{pmatrix} r_d \\ r_{eq} \\ r_{ineq} \\ r_{sz} \end{pmatrix}.$$

However, a full Newton step might be infeasible, because of the positivity constraints on s and z. Therefore, `quadprog` shortens the step, if necessary, to maintain positivity.

Additionally, to maintain a "centered" position in the interior, instead of trying to solve $s_i z_i = 0$, the algorithm takes a positive parameter σ, and tries to solve
$s_i z_i = \sigma r_c$.

`quadprog` replaces r_{sz} in the Newton step equation with $r_{sz} + \Delta s \Delta z - \sigma r_c \mathbf{1}$, where $\mathbf{1}$ is the vector of ones. Also, `quadprog` reorders the Newton equations to obtain a symmetric, more numerically stable system for the predictor step calculation.

For details, see Mehrotra [47].

Multiple Corrections

After calculating the corrected Newton step, `quadprog` can perform more calculations to get both a longer current step, and to prepare for better subsequent steps. These multiple correction calculations can improve both performance and robustness. For details, see Gondzio [62].

Total Relative Error

quadprog calculates a *merit function φ* at every iteration. The merit function is a measure of feasibility, and is also called *total relative error*. quadprog stops if the merit function grows too large. In this case, quadprog declares the problem to be infeasible.

The merit function is related to the KKT conditions for the problem—see "Predictor-Corrector" on page 9-3. Use the following definitions:

$$\rho = \max\left(1, \|H\|, \|\bar{A}\|, \|A_{eq}\|, \|c\|, \|\bar{b}\|, \|b_{eq}\|\right)$$

$$r_{\text{eq}} = A_{\text{eq}}x - b_{\text{eq}}$$

$$r_{\text{ineq}} = \bar{A}x - \bar{b} + s$$

$$r_{\text{d}} = Hx + c + A_{\text{eq}}^T \lambda_{\text{eq}} + \bar{A}^T \bar{\lambda}_{\text{ineq}}$$

$$g = x^T H x + f^T x - \bar{b}^T \bar{\lambda}_{\text{ineq}} - b_{\text{eq}}^T \lambda_{\text{eq}}.$$

The notation $\bar{A}$ and $\bar{b}$ means the linear inequality coefficients, augmented with terms to represent bounds. The notation $\bar{\lambda}_{\text{ineq}}$ similarly represents Lagrange multipliers for the linear inequality constraints, including bound constraints. This was called *z* in "Predictor-Corrector" on page 9-3, and λ_{eq} was called *y*.

The merit function *φ* is

$$\frac{1}{\rho}\left(\max\left(\|r_{\text{eq}}\|_\infty, \|r_{\text{ineq}}\|_\infty, \|r_{\text{d}}\|_\infty\right) + g\right).$$

quadprog iterative display includes a column showing the merit function under the heading Total relative error.

trust-region-reflective 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,

$$\min_{s}\{q(s), \ s \in N\}.$$

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

$$\min\left\{\frac{1}{2}s^T H s + s^T g \ \text{ such that } \ \|Ds\| \leq \Delta\right\},$$

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 9-2 (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 9-2. However, they require time proportional to several factorizations of H. Therefore, for large-scale problems a different approach is needed. Several approximation and heuristic strategies, based on Equation 9-2, 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 9-2 is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The 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

$$H \cdot s_2 = -g,$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$

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 9-2 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 6-2.

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 trust-region methods in Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\},$$

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 9-5 differs from the unconstrained approach in two significant ways. First, an initial feasible point x_0 is computed, using a sparse least-squares step, so that $Ax_0 = b$. Second, Algorithm PCG is replaced with Reduced Preconditioned Conjugate Gradients (RPCG), see [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

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

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

$$\min\{f(x) \text{ such that } l \leq x \leq u\},$$

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 9-7,

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

where

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

and the vector $v(x)$ is defined below, for each $1 \leq i \leq n$:

- 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 9-8 is not differentiable everywhere. Nondifferentiability occurs when $v_i = 0$. You can avoid such points by maintaining strict feasibility, i.e., restricting $l < x < u$.

The scaled modified Newton step s_k for the nonlinear system of equations given by Equation 9-8 is defined as the solution to the linear system

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

at the kth iteration, where

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

and

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

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. $J^v_{ii} = 0$ is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value v_i takes. Further, $|v_i|$ will still be discontinuous at this point, but the function $|v_i| \cdot g_i$ is continuous.

Second, reflections are used to increase the 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 $p^R_i = -p_i$.

active-set quadprog Algorithm

Recall the problem quadprog addresses:

$$\min_x \frac{1}{2}x^T H x + c^T x$$

such that $A\,x \le b$, $Aeq\,x = beq$, and $l \le x \le u$. m is the total number of linear constraints, the sum of number of rows of A and of Aeq.

The `quadprog` `active-set` 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

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

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

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

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c.$$

$\nabla q(p)$ 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

$$Z_k^T H Z_k p = -Z_k^T c.$$

The next step is

$$x_{k+1} = x_k + \alpha d_k, \quad \text{where } d_k = Z_k^T p.$$

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 9-12). 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

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

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

$$S_k^T \lambda_k = c.$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Equation 9-12). 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

$$\min_{\gamma \in \Re,\, x \in \Re^n} \gamma \quad \text{such that}$$
$$A_i x = b_i, \qquad i = 1, \dots, m_e \ \text{(the rows of } Aeq\text{)}$$
$$A_i x - \gamma \le b_i, \quad i = m_e + 1, \dots, m \ \text{(the rows of } A\text{)}.$$

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to Equation 9-20 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):

$$d = -Z_k Z_k^T g_k.$$

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

$$H d_1 = -g_k,$$

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

Quadratic Minimization with Bound Constraints

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

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);
options = optimoptions('quadprog','Algorithm','trust-region-reflective');
[x,fval,exitflag,output] = ...
        quadprog(H,f,[],[],[],[],lb,ub,xstart,options);
```

Looking at the resulting values of exitflag and output,

```
exitflag,output

exitflag =
     3

output =
          algorithm: 'trust-region-reflective'
         iterations: 19
      constrviolation: 0
        firstorderopt: 8.3903e-06
          cgiterations: 1673
               message: 'Optimization terminated: relative function value changing by le…
```

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

```
options = optimoptions(options,'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,output

exitflag =
     3

output =
           algorithm: 'trust-region-reflective'
          iterations: 36
       constrviolation: 0
         firstorderopt: 2.3821e-05
          cgiterations: 1547
               message: 'Optimization terminated: relative function value changing by le…
```

A second strategy would be to use a direct solver at each iteration by setting the
PrecondBandWidth option to inf:

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

Now the number of iterations has dropped to 10:

```
exitflag,output

exitflag =
     3

output =
           algorithm: 'trust-region-reflective'
          iterations: 10
       constrviolation: 0
         firstorderopt: 2.8219e-06
          cgiterations: 0
               message: 'Optimization terminated: relative function value changing by le…
```

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.

You can also use the default `'interior-point-convex'` algorithm to solve this convex problem:

```
options = optimoptions('quadprog','Algorithm','interior-point-convex');
[x,fval,exitflag,output] = ...
        quadprog(H,f,[],[],[],[],lb,ub,[],options);
```

Check the exit flag and output structure:

```
exitflag,output

exitflag =
     1

output =
            message: 'Minimum found that satisfies the constraints.…'
          algorithm: 'interior-point-convex'
       firstorderopt: 1.4120e-06
      constrviolation: 0
           iterations: 8
          cgiterations: []
```

Quadratic Minimization with Dense, Structured Hessian

In this section...
"Take advantage of a structured Hessian" on page 9-19
"Step 1: Decide what part of H to pass to quadprog as the first argument." on page 9-20
"Step 2: Write a function to compute Hessian-matrix products for H." on page 9-20
"Step 3: Call a quadratic minimization routine with a starting point." on page 9-21
"Preconditioning" on page 9-22

Take advantage of a structured Hessian

The `quadprog` trust-region-reflective method can 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 active-set problems and for trust-region-reflective 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 the first part of 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.

The second part of the example shows how to tighten the `TolPCG` tolerance to compensate for an approximate preconditioner instead of an exact H 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 9-22).

```
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 optimoptions 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. Optimization Toolbox software includes the runqpbox4.m file.

```
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 function
```

```
% Choose algorithm and the HessMult option
options = optimoptions(@quadprog,'Algorithm','trust-region-reflective','HessMult',mtxmp

% Pass B to qpbox4mult via the H 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. Then 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,exitflag,output

fval =
```

```
 -1.0538e+03

exitflag =
     3

output =
          algorithm: 'trust-region-reflective'
         iterations: 18
     constrviolation: 0
        firstorderopt: 0.0043
         cgiterations: 30
             message: 'Optimization terminated: relative function value changing by le…
```

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

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

and the first-order optimality is

```
output.firstorderopt
ans =
    0.0043
```

Preconditioning

Sometimes `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 function

% Choose algorithm, the HessMult option, and override the TolPCG option
options = optimoptions(@quadprog,'Algorithm','trust-region-reflective',...
    '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 runqpbox4prec:
        %       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
fval =
-1.0538e+003
```

but the first-order optimality is further reduced.

```
output.firstorderopt
ans =
    0.0028
```

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

Large Sparse Quadratic Program with Interior Point Algorithm

This example shows the value of using sparse arithmetic when you have a sparse problem. The matrix has `n` rows, where you choose `n` to be a large value. A full matrix of size `n`-by-`n` can use up all available memory, but a sparse matrix presents no problem.

The problem is to minimize `x'*H*x/2 + f'*x` subject to

`x(1) + x(2) + ... + x(n) = 0,`

where `f = [-1;-2;-3;...;-n]`.

1 Create the parameter `n` and the utility matrix `T`. The matrix `T` is a sparse circulant matrix that is simply a helper for creating the sparse positive-definite quadratic matrix `H`.

```
n = 30000; % Adjust n to a large value
T = spalloc(n,n,n); % make a sparse circulant matrix
r = 1:n-1;
for m = r
    T(m,m+1)=1;
end
T(n,1) = 1;
```

2 Create a sparse vector `v`. Then create the matrix `H` by shifted versions of `v*v'`. The matrix `T` creates shifts of `v`.

```
v(n) = 0; v(1) = 1; v(2) = 2; v(4) = 3;
v = (sparse(v))';
% Make a banded type of matrix
H = spalloc(n,n,7*n);
r = 1:n;
for m = r
    H = H + v*v';
    v = T*v;
end
```

3 Take a look at the structure of `H`:

```
spy(H)
```
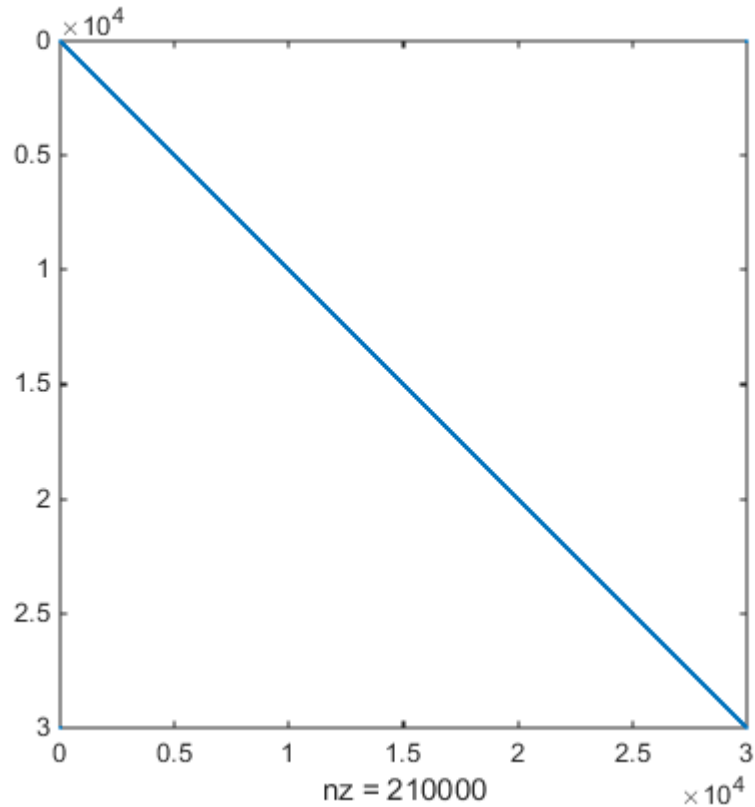
4 Create the problem vector `f` and linear constraint.

```
f = -r; % linear term
A = ones(1,n); b = 0;
```

5 Solve the quadratic programming problem with the `interior-point-convex` algorithm.

```
options = optimoptions(@quadprog,'Algorithm','interior-point-convex');
[x,fval,exitflag,output,lambda] = ...
    quadprog(H,f,A,b,[],[],[],[],[],options);

Minimum found that satisfies the constraints.
Optimization completed because the objective function is
non-decreasing in feasible directions, to within the selected
```

value of the function tolerance, and constraints are satisfied
to within the selected value of the constraint tolerance.

6 View the solution value, output structure, and Lagrange multiplier:

```
fval,output,lambda

fval =
 -3.1331e+10

output =
                 message: 'Minimum found that satisfies the constraints.…'
               algorithm: 'interior-point-convex'
            firstorderopt: 1.1665e-04
          constrviolation: 7.7762e-09
               iterations: 6
             cgiterations: []

lambda =
     ineqlin: 1.5000e+004
       eqlin: [0x1 double]
       lower: [30000x1 double]
       upper: [30000x1 double]
```

Since there are no lower bounds or upper bounds, all the values in `lambda.lower`
and `lambda.upper` are `0`. The inequality constraint is active, since
`lambda.ineqlin` is nonzero.

7 On many computers you cannot create a full n-by-n matrix when n = 30000. So you
can run this problem only using sparse matrices.

```
H2 = zeros(3e4);
```

Out of memory. Type HELP MEMORY for your options.

10

Least Squares

Least-Squares (Model Fitting) Algorithms

In this section...
"Least Squares Definition" on page 10-2
"Trust-Region-Reflective Least Squares" on page 10-3
"Interior-Point Linear Least Squares" on page 10-7
"Levenberg-Marquardt Method" on page 10-7

Least Squares 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 $A\,x \le b$, $Aeq\,x = beq$, $lb \le x \le ub$.

There are several Optimization Toolbox solvers available for various types of $F(x)$ and various types of constraints:

Solver	$F(x)$	Constraints
\	$C\,x - d$	None
lsqnonneg	$C\,x - d$	$x \ge 0$
lsqlin	$C\,x - d$	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
- lsqlin active-set

- lsqlin interior-point
- The algorithm used by lsqnonneg

All the algorithms except the lsqlin active-set algorithm are large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12. 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].

Trust-Region-Reflective Least Squares

Trust-Region-Reflective Least Squares 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,

$$\min_{s}\{q(s),\ s \in N\}.$$

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

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

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 10-2 (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 10-2. However, they require time proportional to several factorizations of H. Therefore, for trust-region problems a different approach is needed. Several approximation and heuristic strategies, based on Equation 10-2, 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 10-2 is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The 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

$$H \cdot s_2 = -g,$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$

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 10-2 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

$$\min_x \sum_i f_i^2(x) = \min_x \|F(x)\|_2^2,$$

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 6-2. However, the structure of the nonlinear least-squares problem is exploited to enhance efficiency. In particular, an approximate Gauss-Newton direction, i.e., a solution s to

$$\min \|Js + F\|_2^2,$$

(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 "Jacobian Multiply Function with Linear Least Squares" on page 10-26.

Interior-Point Linear Least Squares

The `lsqlin` `'interior-point'` algorithm uses the "`interior-point-convex` `quadprog` Algorithm" on page 9-2. The quadprog problem definition is to minimize a quadratic function

$$\min_{x} \frac{1}{2} x^T H x + c^T x$$

subject to linear constraints and bound constraints. The `lsqlin` function minimizes the squared 2-norm of the vector $Cx - d$ subject to linear constraints and bound constraints. In other words, `lsqlin` minimizes

$$
\begin{aligned}
\|Cx - d\|_2^2 &= (Cx - d)^T (Cx - d) \\
&= \left(x^T C^T - d^T \right)(Cx - d) \\
&= \left(x^T C^T C x \right) - \left(x^T C^T d - d^T C x \right) + d^T d \\
&= \frac{1}{2} x^T \left(2 C^T C \right) x + \left(-2 C^T d \right)^T x + d^T d.
\end{aligned}
$$

This fits into the `quadprog` framework by setting the H matrix to $2C^T C$ and the c vector to $(-2C^T d)$. (The additive term $d^T d$ has no effect on the location of the minimum.) After this reformulation of the `lsqlin` problem, the `quadprog` `'interior-point-convex'` algorithm calculates the solution.

Levenberg-Marquardt Method

In the least-squares problem a function $f(x)$ is minimized that is a sum of squares.

$$\min_{x} f(x) = \|F(x)\|_2^2 = \sum_i F_i^2(x).$$

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

$$\min_{x \in \mathfrak{R}^n} \int_{t_1}^{t_2} \left(y(x,t) - \varphi(t) \right)^2 dt,$$

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:

$$\min_{x \in \mathfrak{R}^n} f(x) = \sum_{i=1}^{m} \left(\overline{y}(x,t_i) - \overline{\varphi}(t_i) \right)^2,$$

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

$$F(x) = \begin{bmatrix} \overline{y}(x,t_1) - \overline{\varphi}(t_1) \\ \overline{y}(x,t_2) - \overline{\varphi}(t_2) \\ \dots \\ \overline{y}(x,t_m) - \overline{\varphi}(t_m) \end{bmatrix}.$$

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 6-5, 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

$$G(x) = 2J(x)^T F(x)$$
$$H(x) = 2J(x)^T J(x) + 2Q(x),$$

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:

$$\min_{x \in \Re^n} \|J(x_k) - F(x_k)\|_2^2.$$

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.

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.

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left(J\left(x_k\right)^T J\left(x_k\right) + \lambda_k I \right) d_k = -J\left(x_k\right)^T F\left(x_k\right),$$

or, optionally, of the equations

$$\left(J\left(x_k\right)^T J\left(x_k\right) + \lambda_k diag\left(J\left(x_k\right)^T J\left(x_k\right)\right) \right) d_k = -J\left(x_k\right)^T F\left(x_k\right),$$

where the scalar λ_k controls both the magnitude and direction of d_k. Set option ScaleProblem to 'none' to choose Equation 10-12, and set ScaleProblem to 'Jacobian' to choose Equation 10-13.

You set the initial value of the parameter λ_0 using the InitDamping option. Occasionally, the 0.01 default value of this option can be unsuitable. If you find that the

Levenberg-Marquardt algorithm makes little initial progress, try setting `InitDamping` to a different value than the default, perhaps `1e2`.

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. When the step is successful (gives a lower function value), the algorithm sets $\lambda_{k+1} = \lambda_k/10$. When the step is unsuccessful, the algorithm sets $\lambda_{k+1} = \lambda_k*10$.

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 10-1, 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.
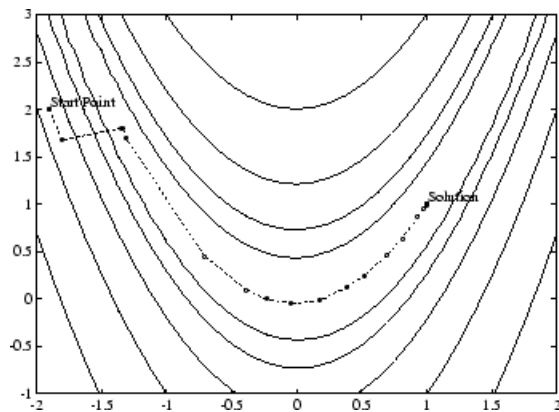
Figure 10-1. Levenberg-Marquardt Method on Rosenbrock's Function

For an animated version of this figure, enter `bandem` at the MATLAB command line.

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/optimdemos` folder. 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.
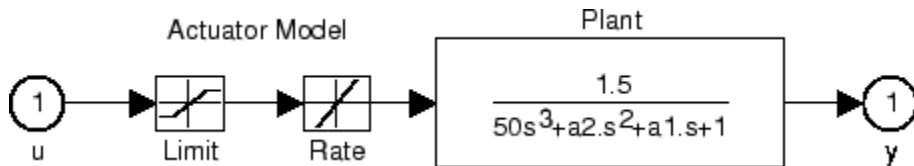
Plant with Actuator Saturation

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. You can see this response by opening the model (type optsim at the command line or click the model name), and selecting **Run** from the **Simulation** menu. The response plots to the scope.
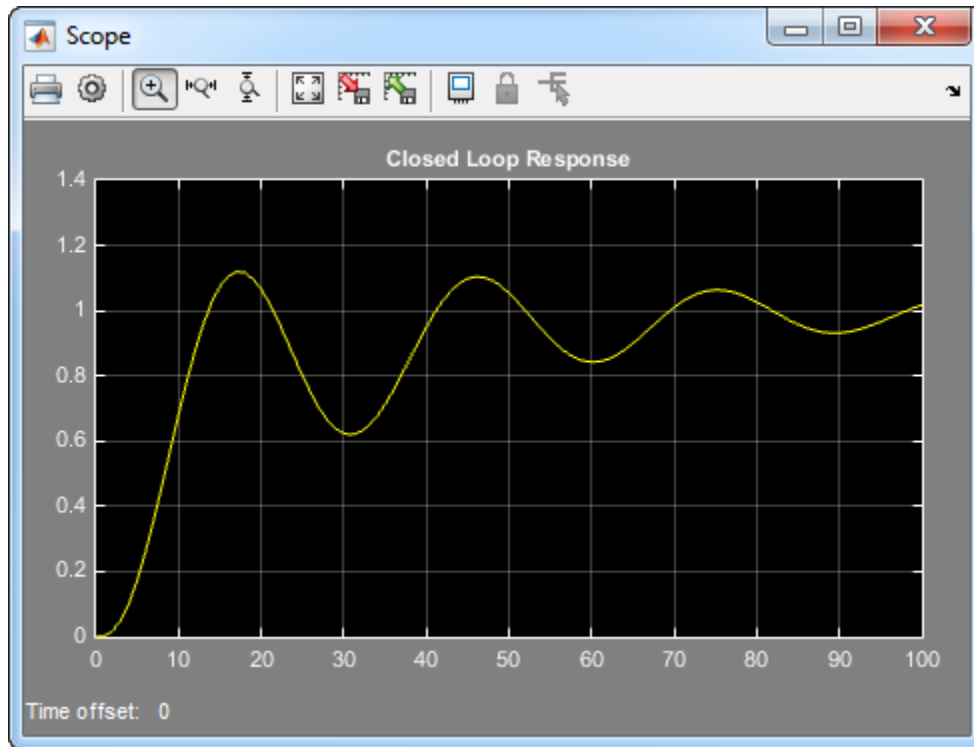
Closed-Loop Response

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.

Tunable Variables are PID gains, Kp, Ki, and Kd.
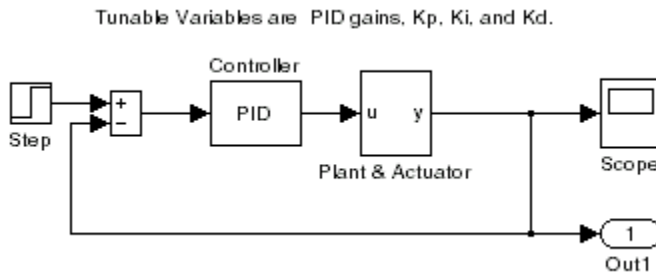
Closed-Loop Model

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 the function `tracklsq`, which returns the error signal `yout`, the output computed by calling `sim`, minus the input signal `1`. The code for `tracklsq` is contained in the file `runtracklsq.m`, shown below.

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 `'levenberg-marquardt'` 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` runs 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 `SrcWorkspace` option is set to `'Current'` to tell `sim` to run the simulation in the current workspace. The simulation is performed to `100` seconds.

When the simulation is completed, the `myobj` object is created in the current workspace (that is, the workspace of `tracklsq`). The Outport block in the block diagram model puts the `yout` field of the object 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 = optimoptions(@lsqnonlin,'Algorithm','levenberg-marquardt',...
    'Display','off','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);

        % Set sim options and compute function value
        myobj = sim('optsim','SrcWorkspace','Current', ...
            'StopTime','100');
        F = myobj.get('yout') - 1;
    end
end
```

Copy the code for `runtracklsq` to a file named `runtracklsq.m`, placed in a folder on your MATLAB path.

When you run `runtracklsq`, the optimization gives the solution for the proportional, integral, and derivative (Kp, Ki, Kd) gains of the controller:

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

Kp =
    2.9633

Ki =
```

```
    0.1436

Kd =
   13.1386
```

Here is the resulting closed-loop step response.
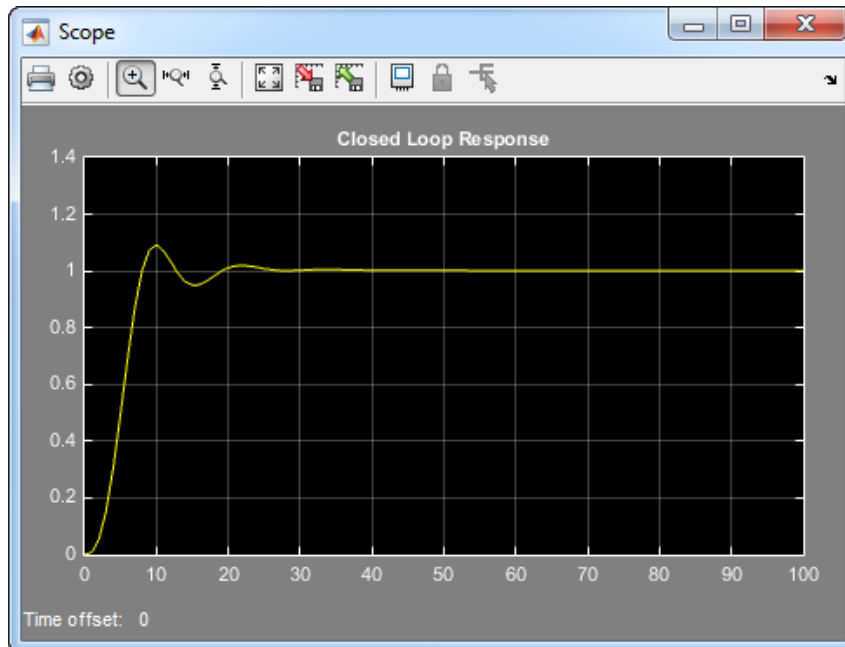
Closed-Loop Response Using lsqnonlin

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 Design 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.

Nonlinear Least Squares With and Without Jacobian

In this section...
"Problem definition and solution technique" on page 10-17
"Step 1: Write a file myfun.m that computes the objective function values." on page 10-17
"Step 2: Call the nonlinear least-squares routine." on page 10-18
"Step 3: Include a Jacobian." on page 10-18

Problem definition and solution technique

This example shows how to solve a nonlinear least squares problem in two ways. It first shows the solution without using a Jacobian function. Then it shows how to include a Jacobian, and it shows the efficiency improvement that the Jacobian gives.

The problem has 10 terms with 2 unknowns: find x, a two-dimensional vector, that minimizes

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

starting at the point $x0 = [0.3,0.4]$.

Because `lsqnonlin` assumes that the sum of squares is not explicitly formed in the user function, the function passed to `lsqnonlin` should 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 a 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,res,eflag,output1] = 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 initially, it detects that `Jstr` is actually a dense matrix, i.e., 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 72 function evaluations, this example gives the solution

```
x,resnorm

x =
    0.2578    0.2578

resnorm =
    124.3622
```

Most computer systems can handle much larger full problems, say into the hundreds 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 always runs faster if this information is provided.

Step 3: Include a Jacobian.

The objective function is simple enough to calculate its Jacobian. Following the definition in "Jacobians of Vector Functions" on page 2-25, a Jacobian function represents the matrix

$$J_{kj}(x) = \frac{\partial F_k(x)}{\partial x_j}.$$

Here, $F_k(x$ is the kth component of the objective function. This example has

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2},$$

so

$$J_{k1}(x) = -ke^{kx_1}$$

$$J_{k2}(x) = -ke^{kx_2}.$$

Modify the objective function file.

```
function [F,J] = myfun(x)
k = 1:10;
F = 2 + 2*k-exp(k*x(1))-exp(k*x(2));
if nargout > 1
    J = zeros(10,2);
    J(k,1) = -k.*exp(k*x(1));
    J(k,2) = -k.*exp(k*x(2));
end
```

Set options so the solver uses the Jacobian.

```
opts = optimoptions(@lsqnonlin,'Jacobian','on');
```

Run the solver.

```
x0 = [0.3 0.4]; % Starting guess
[x,resnorm,res,eflag,output2] = lsqnonlin(@myfun,x0,[],[],opts);
```

The solution is the same as before.

```
x,resnorm

x =
    0.2578    0.2578

resnorm =
    124.3622
```

The advantage to using a Jacobian is that the solver takes fewer function evaluations, 24 instead of 72.

```
[output1.funcCount,output2.funcCount]
```

```
ans =
   72    24
```

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,resnorm,output

exitflag =
     3

resnorm =
   22.5794

output =
       iterations: 10
        algorithm: 'trust-region-reflective'
     firstorderopt: 2.7870e-05
      cgiterations: 42
           message: 'Optimization terminated: relative function value changing by less…
```

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 6-25, 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 = optimoptions('lsqlin','PrecondBandWidth',inf);
[x,resnorm,residual,exitflag,output] = ...
            lsqlin(C,d,[],[],[],[],lb,[],[],options);
```

The first-order optimality measure decreases:

```
exitflag,resnorm,output

exitflag =
     1

resnorm =
   22.5794

output =
       iterations: 12
        algorithm: 'trust-region-reflective'
    firstorderopt: 5.5907e-15
     cgiterations: O
          message: 'Optimization terminated: first order optimality with optimality…'
```

Optimization App with the lsqlin Solver

In this section...

The Problem

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

Note: The Optimization app warns that it will be removed in a future release.

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:

$$\min_{x} f(x) = x_1^2 + x_2^2 + x_3^2,$$
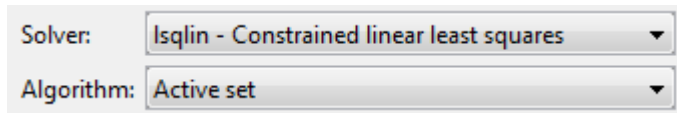
subject to the constraint
$x_1 + 2x_2 + 4x_3 = 7$.

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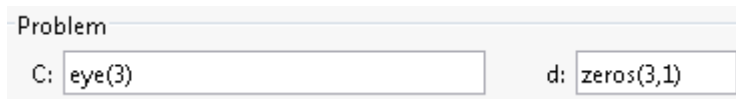
1 Enter `optimtool` in the Command Window to open the Optimization app.

2 Select `lsqlin` from the selection of solvers. Use the `Active set` algorithm.

| Solver: | lsqlin - Constrained linear least squares | ▼ |
| Algorithm: | Active set | ▼ |

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.
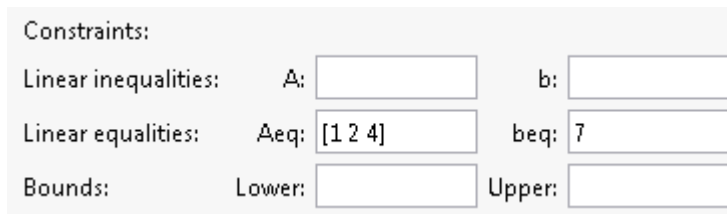
Problem

C: eye(3)   d: zeros(3,1)

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.
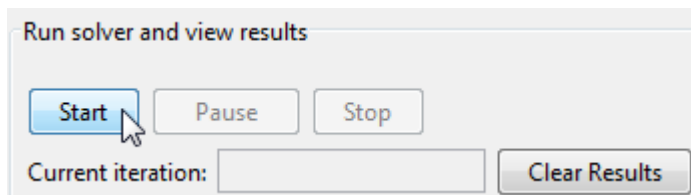
Constraints:

Linear inequalities:	A:		b:	
Linear equalities:	Aeq: [1 2 4]		beq: 7	
Bounds:	Lower:		Upper:	

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

Run solver and view results

Start   Pause   Stop

Current iteration: Clear Results

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
  ```

Jacobian Multiply Function with Linear Least Squares

You can solve a least-squares problem of the form

$$\min_{x} \frac{1}{2} \| C \cdot x - d \|_2^2$$

such that $A x \leq b$, $Aeq\ x = beq$, $lb \leq x \leq ub$, 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 $(-1)^{k+1}/k$:
$v = [1, -1/2, 1/3, -1/4, \dots, -1/n]$,

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
$d = [n - 1; n - 2; \dots; -n]$,

and the constraints are $-5 \leq x(i) \leq 5$ for $i = 1, \dots, n$.

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 ⟹ w = C*Y
- flag < 0 ⟹ w = C'*Y
- flag = 0 ⟹ 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 = optimoptions('lsqlin','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
```

10-29

```
output
output =
      iterations: 16
       algorithm: 'trust-region-reflective'
   firstorderopt: 7.5143e-05
    cgiterations: 36
         message: 'Optimization terminated: relative function value changing by les…'
```

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:

```
rng(5489,'twister') % reproducible
xdata = -2*log(rand(100,1));
ydata = (ones(100,1) + .1*randn(100,1)) + (3*ones(100,1)+...
    0.5*randn(100,1)).*exp((-(2*ones(100,1)+...
    .5*randn(100,1))).*xdata);
```

The modeled relationship between xdata and ydata is

$$ydata_i = a_1 + a_2 \exp(-a_3 xdata_i) + \varepsilon_i.$$

The script generates xdata from 100 independent samples from an exponential distribution with mean 2. It generates ydata from Equation 10-14 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);

Local minimum possible.
```

```
lsqcurvefit stopped because the final change in the
sum of squares relative to its initial value is
less than the default value of the function tolerance.
```

To see the resulting least-squares estimate of $\hat{a}$, enter:

```
ahat

ahat =
    1.0169
    3.1444
    2.1596
```

The fitted values `ahat` are within 8% of `a = [1;3;2]`.

If you have Statistics and Machine Learning Toolbox™ software, use the `nlparci` function to generate confidence intervals for the `ahat` estimate.

Fit a Model to Complex-Valued Data

This example shows how to perform nonlinear fitting of complex-valued data. While most Optimization Toolbox solvers and algorithms operate only on real-valued data, the `levenberg-marquardt` algorithm works on both real-valued and complex-valued data.

Do not set the `FunValCheck` option to `'on'` when using complex data. The solver errors.

Data Model

The data model is a simple exponential:

$$y(x) = v_1 + v_2 e^{v_3 x}.$$

The x is input data, y is the response, and v is a complex-valued vector of coefficients. The goal is to estimate v from x and noisy observations y.

Artificial Data with Noise

Generate artificial data for the model. Take the complex coefficient vector v as `[2;3+4i;-.5+.4i]`. Take the observations x as exponentially distributed. Add complex-valued noise to the responses y.

```
rng default % for reproducibility
N = 100; % number of observations
v0 = [2;3+4i;-.5+.4i]; % coefficient vector
xdata = -log(rand(N,1)); % exponentially distributed
noisedata = randn(N,1).*exp((1i*randn(N,1))); % complex noise
cplxydata = v0(1) + v0(2).*exp(v0(3)*xdata) + noisedata;
```

Fit the Model to Recover the Coefficient Vector

The difference between the response predicted by the data model and an observation (`xdata` for x and response `cplxydata` for y) is:

```
objfcn = @(v)v(1)+v(2)*exp(v(3)*xdata) - cplxydata;
```

Use either `lsqnonlin` or `lsqcurvefit` to fit the model to the data. This example first uses `lsqnonlin`. Because the data is complex, set the `Algorithm` option to `'levenberg-marquardt'`.

```
opts = optimoptions(@lsqnonlin,...
```

```
      'Algorithm','levenberg-marquardt','Display','off');
x0 = (1+1i)*[1;1;1]; % arbitrary initial guess
[vestimated,resnorm,residuals,exitflag,output] = lsqnonlin(objfcn,x0,[],[],opts);
vestimated,resnorm,exitflag,output.firstorderopt

vestimated =

   2.1581 + 0.1351i
   2.7399 + 3.8012i
  -0.5338 + 0.4660i


resnorm =

  100.9933


exitflag =

     3


ans =

    0.0013
```

lsqnonlin recovers the complex coefficient vector to about one significant digit. The norm of the residual is sizable, indicating that the noise keeps the model from fitting all the observations. The exit flag is 3, not the preferable 1, because the first-order optimality measure is about 1e-3, not below 1e-6.

Alternative: Use lsqcurvefit

To fit using lsqcurvefit, write the model to give just the responses, not the responses minus the response data.

```
objfcn = @(v,xdata)v(1)+v(2)*exp(v(3)*xdata);
```

Use lsqcurvefit options and syntax.

```
opts = optimoptions(@lsqcurvefit,opts); % reuse the options
[vestimated,resnorm] = lsqcurvefit(objfcn,x0,xdata,cplxydata,[],[],opts)

vestimated =
```

```
   2.1581 + 0.1351i
   2.7399 + 3.8012i
  -0.5338 + 0.4660i


resnorm =

  100.9933
```

The results match those from `lsqnonlin`, because the underlying algorithms are identical. Use whichever solver you find more convenient.

Alternative: Split Real and Imaginary Parts

To use the `trust-region-reflective` algorithm, such as when you want to include bounds, you must split the real and complex parts of the coefficients into separate variables. For this problem, split the coefficients as follows:

$$
\begin{aligned}
y &= v_1 + iv_2 + (v_3 + iv_4)\exp\big((v_5 + iv_6)x\big) \\
&= \big(v_1 + v_3\exp(v_5 x)\cos(v_6 x) - v_4\exp(v_5 x)\sin(v_6 x)\big) \\
&\quad + i\big(v_2 + v_4\exp(v_5 x)\cos(v_6 x) + v_3\exp(v_5 x)\sin(v_6 x)\big).
\end{aligned}
$$

Write the response function for `lsqcurvefit`.

```
function yout = cplxreal(v,xdata)

yout = zeros(length(xdata),2); % allocate yout
expcoef = exp(v(5)*xdata(:)); % magnitude
coscoef = cos(v(6)*xdata(:)); % real cosine term
sincoef = sin(v(6)*xdata(:)); % imaginary sin term
yout(:,1) = v(1) + expcoef.*(v(3)*coscoef - v(4)*sincoef);
yout(:,2) = v(2) + expcoef.*(v(4)*coscoef + v(3)*sincoef);
```

Save this code as the file `cplxreal.m` on your MATLAB path.

Split the response data into its real and imaginary parts.

```
ydata2 = [real(cplxydata),imag(cplxydata)];
```

The coefficient vector `v` now has six dimensions. Initialize it as all ones, and solve the problem using `lsqcurvefit`.

```
x0 = ones(6,1);
```

```
[vestimated,resnorm,residuals,exitflag,output] = ...
    lsqcurvefit(@cplxreal,x0,xdata,ydata2);
vestimated,resnorm,exitflag,output.firstorderopt

vestimated =

    2.1582
    0.1351
    2.7399
    3.8012
   -0.5338
    0.4660


resnorm =

  100.9933


exitflag =

     3


ans =

    0.0018
```

Interpret the six-element vector `vestimated` as a three-element complex vector, and you see that the solution is virtually the same as the previous solutions.

11

Systems of Equations

Equation Solving Algorithms

In this section...

Equation Solving 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 $F_i(x) = 0$.

`fsolve` attempts to solve systems of equations by minimizing the sum of squares of the components. If the sum of squares is zero, the system of equation is solved. `fsolve` has three algorithms:

- Trust-region-reflective
- Trust-region dogleg
- Levenberg-Marquardt

All algorithms are large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12.

The `fzero` function solves a single one-dimensional equation.

The \ function solves systems of linear equations.

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,

$$\min_{s}\{q(s),\ s \in N\}.$$

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

$$\min\left\{\frac{1}{2}s^T H s + s^T g \ \text{ such that } \ \|Ds\| \leq \Delta\right\},$$

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 11-2 (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 11-2. However, they require time proportional to several factorizations of H. Therefore, for trust-region problems a

different approach is needed. Several approximation and heuristic strategies, based on Equation 11-2, 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 11-2 is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The 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

$$H \cdot s_2 = -g,$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$

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 11-2 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., $d^THd \leq 0$. 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 6-2.

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 6-2) improves robustness when starting far from the solution and

handles the case when $J(x_k)$ is singular. To use a trust-region strategy, a merit function is needed to decide if x_{k+1} is better or worse than x_k. A possible choice is

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

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

The Newton step d_k is a root of
$M(x_k + d) = F(x_k) + J(x_k)d,$

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

$$\min_d m(d) = \frac{1}{2} \left\| M(x_k + d) \right\|_2^2 = \frac{1}{2} \left\| F(x_k) + J(x_k)d \right\|_2^2$$
$$= \frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T J(x_k)^T J(x_k)d.$$

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

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

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 11-6. 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 11-5.

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.

Levenberg-Marquardt Method

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left(J\left(x_k\right)^T J\left(x_k\right) + \lambda_k I\right)d_k = -J\left(x_k\right)^T F\left(x_k\right),$$

or, optionally, of the equations

$$\left(J\left(x_k\right)^T J\left(x_k\right) + \lambda_k diag\left(J\left(x_k\right)^T J\left(x_k\right)\right)\right)d_k = -J\left(x_k\right)^T F\left(x_k\right),$$

where the scalar λ_k controls both the magnitude and direction of d_k. Set option ScaleProblem to 'none' to choose Equation 11-7, and set ScaleProblem to 'Jacobian' to choose Equation 11-8.

When λ_k is zero, the direction d_k is 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.

\ 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.

Nonlinear Equations with Analytic Jacobian

This example demonstrates the use of the default trust-region-dogleg `fsolve` algorithm (see "Large-Scale vs. Medium-Scale Algorithms" on page 2-12). 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:

$$F(1) = 1 - x_1$$

$$F(2) = 10\left(x_2 - x_1^2\right)$$

$$F(3) = 1 - x_3$$

$$F(4) = 10\left(x_4 - x_3^2\right)$$

$$\vdots$$

$$F(n-1) = 1 - x_{n-1}$$

$$F(n) = 10\left(x_n - x_{n-1}^2\right).$$

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

Step 1: Write a 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 = optimoptions(@fsolve,'Display','iter','Jacobian','on');
```

```
[x,F,exitflag,output,JAC] = fsolve(@bananaobj,xO,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        8563.84                615              1
    1           2        3093.71       1        329              1
    2           3        225.104     2.5        34.8             2.5
    3           4        212.48      6.25       34.1             6.25
    4           5        212.48      6.25       34.1             6.25
    5           6        102.771    1.5625      6.39             1.56
    6           7        102.771    3.90625     6.39             3.91
    7           8        87.7443    0.976563    2.19             0.977
    8           9        74.1426    2.44141     6.27             2.44
    9          10        74.1426    2.44141     6.27             2.44
   10          11        52.497     0.610352    1.52             0.61
   11          12        41.3297    1.52588     4.63             1.53
   12          13        34.5115    1.52588     6.97             1.53
   13          14        16.9716    1.52588     4.69             1.53
   14          15        8.16797    1.52588     3.77             1.53
   15          16        3.55178    1.52588     3.56             1.53
   16          17        1.38476    1.52588     3.31             1.53
   17          18        0.219553   1.16206     1.66             1.53
   18          19        0          0.0468565   0                1.53
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

Nonlinear Equations with Finite-Difference Jacobian

In the example "Nonlinear Equations with Analytic Jacobian" on page 11-9, 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 `optimoptions`.

This example uses `bananaobj` from the example "Nonlinear Equations with Analytic Jacobian" on page 11-9 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 = optimoptions(@fsolve,'Display','iter','Jacobian','off');
[x,F,exitflag,output,JAC] = fsolve(@bananaobj,x0,options);
```

The example produces the following output:

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	65	8563.84		615	1
1	130	3093.71	1	329	1
2	195	225.104	2.5	34.8	2.5
3	260	212.48	6.25	34.1	6.25
4	261	212.48	6.25	34.1	6.25
5	326	102.771	1.5625	6.39	1.56
6	327	102.771	3.90625	6.39	3.91
7	392	87.7443	0.976562	2.19	0.977
8	457	74.1426	2.44141	6.27	2.44
9	458	74.1426	2.44141	6.27	2.44
10	523	52.497	0.610352	1.52	0.61
11	588	41.3297	1.52588	4.63	1.53
12	653	34.5115	1.52588	6.97	1.53
13	718	16.9716	1.52588	4.69	1.53
14	783	8.16797	1.52588	3.77	1.53
15	848	3.55178	1.52588	3.56	1.53
16	913	1.38476	1.52588	3.31	1.53
17	978	0.219553	1.16206	1.66	1.53
18	1043	0	0.0468565	0	1.53

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

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.

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 2-12).

Step 1: Write a 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 = optimoptions(@fsolve,'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
     0          1          1011                        19
     1          2        16.1942     7.91898          2.35
     2          3       0.0228027    1.33142          0.291
     3          4      0.000103359  0.0433329         0.0201
     4          5       7.3792e-07   0.0022606        0.000946
     5          6      4.02299e-10  0.000268381       4.12e-05

Equation solved, inaccuracy possible.

The vector of function values is near zero, as measured by the default value
of the function tolerance. However, the last step was ineffective.
```

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 = optimoptions(@fsolve,'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
    0          1           1011                            19
    1          2         16.0839        7.92496           1.92
    2          3        0.0458181        1.3279           0.579
    3          4       0.000101184      0.0631898         0.0203
    4          5       3.16615e-07      0.00273698        0.00079
    5          6       9.72481e-10      0.00018111        5.82e-05

Equation solved, inaccuracy possible.

The vector of function values is near zero, as measured by the default value
of the function tolerance. However, the last step was ineffective.
```

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 6-23.

Setting `PrecondBandWidth` to `Inf` (this is the default) means that the solver uses Cholesky factorization rather than PCG.

Nonlinear Equations with Jacobian Sparsity Pattern

In the example "Nonlinear Equations with Analytic Jacobian" on page 11-9, 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 a sparse matrix Jstr 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 with 1000 variables, 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 nlsf1 without the Jacobian. Sparse finite-differencing is used to estimate the sparse Jacobian matrix as needed.

Step 1: Write a 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 = optimoptions(@fsolve,'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
     0          5           1011                      19
     1         10         16.0839     7.92496         1.92
     2         15        0.0458179     1.3279         0.579
     3         20       0.000101184   0.0631896       0.0203
     4         25       3.16616e-07   0.00273698      0.00079
     5         30       9.72483e-10   0.00018111      5.82e-05
```

```
Equation solved, inaccuracy possible.

The vector of function values is near zero, as measured by the default value
of the function tolerance. However, the last step was ineffective.
```

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 = optimoptions(@fsolve,'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
     0          5           1011                      19
     1         10         15.9018     7.92421         1.89
     2         15        0.0128161     1.32542        0.0746
     3         20       1.73502e-08    0.0397923      0.000196
     4         25       1.10716e-18    4.55495e-05    2.74e-09
```

```
Equation solved.
```

```
fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

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.

Nonlinear Systems with Constraints

In this section...

Solve Equations with Inequality Constraints

`fsolve` solves systems of nonlinear equations. However, it does not allow you to include any constraints, even bound constraints. The question is, how can you solve systems of nonlinear equations when you have constraints?

The short answer is, there are no guarantees that a solution exists that satisfies your constraints. There is no guarantee that any solution exists, even one that does not satisfy your constraints. Nevertheless, there are techniques that can help you search for solutions that satisfy your constraints.

To illustrate the techniques, consider how to solve the equations

$$F_1(x) = (x_1 + 1)(10 - x_1)\frac{1 + x_2^2}{1 + x_2^2 + x_2}$$

$$F_2(x) = (x_2 + 2)(20 - x_2)\frac{1 + x_1^2}{1 + x_1^2 + x_1},$$

where the components of x must be nonnegative. Clearly, there are four solutions to the equations:
$x = (-1, -2)$
$x = (10, -2)$,
$x = (-1, 20)$,
$x = (10, 20)$.

There is only one solution that satisfies the constraints, namely $x = (10, 20)$.

To solve the equations numerically, first enter code to calculate $F(x)$.

```
function F = fbnd(x)

F(1) = (x(1)+1)*(10-x(1))*(1+x(2)^2)/(1+x(2)^2+x(2));
F(2) = (x(2)+2)*(20-x(2))*(1+x(1)^2)/(1+x(1)^2+x(1));
```

Save this code as the file `fbnd.m` on your MATLAB path.

Use Different Start Points

Generally, a system of *N* equations in *N* variables has isolated solutions, meaning each solution has no nearby neighbors that are also solutions. So one way to search for a solution that satisfies some constraints is to generate a number of initial points `x0`, and run `fsolve` starting at each `x0`.

For this example, to look for a solution to Equation 11-9, take 10 random points that are normally distributed with mean 0 and standard deviation 100.

```
rng default % for reproducibility
N = 10; % try 10 random start points
pts = 100*randn(N,2); % initial points are rows in pts
soln = zeros(N,2); % allocate solution
opts = optimoptions('fsolve','Display','off');
for k = 1:N
    soln(k,:) = fsolve(@fbnd,pts(k,:),opts); % find solutions
end
```

Examine the solutions in `soln`, and you find several that satisfy the constraints.

Use Different Algorithms

There are three `fsolve` algorithms. Each can lead to different solutions.

For this example, take `x0 = [1,9]` and examine the solution each algorithm returns.

```
x0 = [1,9];
opts = optimoptions(@fsolve,'Display','off',...
    'Algorithm','trust-region-dogleg');
x1 = fsolve(@fbnd,x0,opts)

x1 =
```

```
   -1.0000   -2.0000

opts.Algorithm = 'trust-region-reflective';
x2 = fsolve(@fbnd,x0,opts)

x2 =

   -1.0000   20.0000

opts.Algorithm = 'levenberg-marquardt';
x3 = fsolve(@fbnd,x0,opts)

x3 =

    0.9523    8.9941
```

Here, all three algorithms find different solutions for the same initial point. In fact, `x3` is not even a solution, but is simply a locally stationary point.

Use lsqnonlin with Bounds

`lsqnonlin` tries to minimize the sum of squares of the components of a vector function $F(x)$. Therefore, it attempts to solve the equation $F(x) = 0$. Furthermore, `lsqnonlin` accepts bound constraints.

Formulate the example problem for `lsqnonlin` and solve it.

```
lb = [0,0];
rng default
x0 = 100*randn(2,1);
[x res] = lsqnonlin(@fbnd,x0,lb)

x =

   10.0000
   20.0000


res =

   2.4783e-25
```

You can use `lsqnonlin` with the Global Optimization Toolbox `MultiStart` solver to search over many initial points automatically. See "MultiStart Using lsqcurvefit or lsqnonlin".

Set Equations and Inequalities as fmincon Constraints

You can reformulate the problem and use `fmincon` as follows:

- Give a constant objective function, such as `@(x)0`, which evaluates to `0` for each `x`.
- Set the `fsolve` objective function as the nonlinear equality constraints in `fmincon`.
- Give any other constraints in the usual `fmincon` syntax.

For this example, write a function file for the nonlinear inequality constraint.

```
function [c,ceq] = fminconstr(x)

c = []; % no nonlinear inequality
ceq = fbnd(x); % the fsolve objective is fmincon constraints
```

Save this code as the file `fminconstr.m` on your MATLAB path.

Solve the constrained problem.

```
lb = [0,0]; % lower bound constraint
rng default % reproducible initial point
x0 = 100*randn(2,1);
opts = optimoptions(@fmincon,'Algorithm','interior-point','Display','off');
x = fmincon(@(x)0,x0,[],[],[],[],lb,[],@fminconstr,opts)

x =

   10.0000
   20.0000
```

12

Parallel Computing for Optimization

Parallel Computing in Optimization Toolbox Functions

In this section...
"Parallel Optimization Functionality" on page 12-2
"Parallel Estimation of Gradients" on page 12-2
"Nested Parallel Functions" on page 12-3

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; just don't set them both to `'on'`.
- Parallel computing is enabled with `parpool`, a Parallel Computing Toolbox function.
- The option `UseParallel` is set to `true`. The default value of this option is `false`.

When these conditions hold, the solvers compute estimated gradients in parallel.

Note: Even when running in parallel, a solver occasionally calls the objective and nonlinear constraint functions serially on the host machine. Therefore, ensure that your functions have no assumptions about whether they are evaluated in serial or 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

$$\nabla f(x) \approx \left[\frac{f(x + \Delta_1 e_1) - f(x)}{\Delta_1}, \frac{f(x + \Delta_2 e_2) - f(x)}{\Delta_2}, \dots, \frac{f(x + \Delta_n e_n) - f(x)}{\Delta_n} \right],$$

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.

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

$$\nabla f(x) \approx \left[\frac{f(x + \Delta_1 e_1) - f(x - \Delta_1 e_1)}{2\Delta_1}, \dots, \frac{f(x + \Delta_n e_n) - f(x - \Delta_n e_n)}{2\Delta_n} \right].$$

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 `optimoptions` 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 12-2, are satisfied. When parfor Runs In Parallel 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

Only the outermost parfor loop
runs in parallel

(1)
```
...
parfor (i=1:10)
   x(i)=fmincon(@userfcn,...)
   ...
end
```

If UseParallel = 'always'
fmincon runs in parallel

(2)
```
...
for (i=1:10)
   x(i)=fmincon(@userfcn,...)
   ...
end
```

If UseParallel = 'never'
userfcn can use parfor in parallel

(3)
```
...
for (i=1:10)
   x(i)=fmincon(@userfcn,...)
   ...
end
```

When parfor Runs In Parallel

Using Parallel Computing in Optimization Toolbox

In this section...

Using Parallel Computing with Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a parallel pool of several workers with a Parallel Computing Toolbox license. For a description of Parallel Computing Toolbox software, see "Getting Started with Parallel Computing Toolbox".

Suppose you have a dual-core processor, and want to use parallel computing:

• Enter

```
parpool
```
at the command line. MATLAB starts a pool of workers using the multicore processor. If you had previously set a nondefault cluster profile, you can enforce multicore (local) computing:

```
parpool('local')
```

Note: Depending on your preferences, MATLAB can start a parallel pool automatically. To enable this feature, check **Automatically create a parallel pool** in **Home > Parallel > Parallel Preferences**.

• • For command-line use, enter

```
options = optimoptions('solvername','UseParallel',true);
```
  • For Optimization app, check **Options > Approximated derivatives > Evaluate in parallel**.

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcp)
```

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   Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation.

To perform a basic check:

a   At the command line, enter

```
parpool(prof)
```
where `prof` is your cluster profile.

b   Workers must be able to access your objective function file and, if applicable, your nonlinear constraint function file. There are two ways of ensuring access:

i   Distribute the files to the workers using the `parpool` `AttachedFiles` argument. For example, if `objfun.m` is your objective function file, and `constrfun.m` is your nonlinear constraint function file, enter

```
parpool('AttachedFiles',{'objfun.m','constrfun.m'});
```

Workers access their own copies of the files.

ii   Give a network file path to your files. If *network_file_path* is the network path to your objective or constraint function files, enter

```
pctRunOnAll('addpath network_file_path')
```

Workers access the function files over the network.

c   Check whether a file is on the path of every worker by entering

```
pctRunOnAll('which filename')
```
If any worker does not have a path to the file, it reports

```
filename not found.
```

2   •   For command-line use, enter

```
options = optimoptions('solvername','UseParallel',true);
```

- For Optimization app, check **Options > Approximated derivatives > Evaluate in parallel**.

After you establish your parallel computing environment, applicable solvers automatically use parallel computing whenever you call them with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcp)
```

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 `true`, and ensure that there is no parallel pool using `delete(gcp)`. Uncheck **Automatically create a parallel pool** in **Home > Parallel > Parallel Preferences** so MATLAB does not create a parallel pool . 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 `true`, and create a parallel pool using `parpool`. 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.

Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™

This example shows how to how to speed up the minimization of an expensive optimization problem using functions in Optimization Toolbox™ and Global Optimization Toolbox. In the first part of the example we solve the optimization problem by evaluating functions in a serial fashion and in the second part of the example we solve the same problem using the parallel for loop (`parfor`) feature by evaluating functions in parallel. We compare the time taken by the optimization function in both cases.

Expensive Optimization Problem

For the purpose of this example, we solve a problem in four variables, where the objective and constraint functions are made artificially expensive by pausing.

```
type expensive_objfun.m
type expensive_confun.m
```

```
function f = expensive_objfun(x)
%EXPENSIVE_OBJFUN An expensive objective function used in optimparfor example.

%   Copyright 2007-2014 The MathWorks, Inc.
%   $Revision: 1.1.8.2 $  $Date: 2013/05/04 00:47:14 $

% Simulate an expensive function by pausing
pause(0.1)
% Evaluate objective function
f = exp(x(1)) * (4*x(3)^2 + 2*x(4)^2 + 4*x(1)*x(2) + 2*x(2) + 1);

function [c,ceq] = expensive_confun(x)
%EXPENSIVE_CONFUN An expensive constraint function used in optimparfor example.

%   Copyright 2007-2014 The MathWorks, Inc.
%   $Revision: 1.1.8.2 $  $Date: 2013/05/04 00:47:13 $

% Simulate an expensive function by pausing
pause(0.1);
% Evaluate constraints
c = [1.5 + x(1)*x(2)*x(3) - x(1) - x(2) - x(4);
     -x(1)*x(2) + x(4) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Minimizing Using `fmincon`

We are interested in measuring the time taken by `fmincon` in serial so that we can compare it to the parallel `fmincon` evaluation.

```
startPoint = [-1 1 1 -1];
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
startTime = tic;
xsol = fmincon(@expensive_objfun,startPoint,[],[],[],[],[],[],@expensive_confun,options
time_fmincon_sequential = toc(startTime);
fprintf('Serial FMINCON optimization takes %g seconds.\n',time_fmincon_sequential);
```

```
                                                             Norm of First-order
 Iter F-count            f(x)  Feasibility   Steplength       step   optimality
    0       5    1.839397e+00   1.500e+00                            3.311e+00
    1      12   -8.841073e-01   4.019e+00    4.900e-01    2.335e+00   7.015e-01
    2      17   -1.382832e+00   0.000e+00    1.000e+00    1.142e+00   9.272e-01
    3      22   -2.241952e+00   0.000e+00    1.000e+00    2.447e+00   1.481e+00
    4      27   -3.145762e+00   0.000e+00    1.000e+00    1.756e+00   5.464e+00
    5      32   -5.277523e+00   6.413e+00    1.000e+00    2.224e+00   1.357e+00
    6      37   -6.310709e+00   0.000e+00    1.000e+00    1.099e+00   1.309e+00
    7      43   -6.447956e+00   0.000e+00    7.000e-01    2.191e+00   3.631e+00
    8      48   -7.135133e+00   0.000e+00    1.000e+00    3.719e-01   1.205e-01
    9      53   -7.162732e+00   0.000e+00    1.000e+00    4.083e-01   2.935e-01
   10      58   -7.178390e+00   0.000e+00    1.000e+00    1.591e-01   3.110e-02
   11      63   -7.180399e+00   1.191e-05    1.000e+00    2.644e-02   1.553e-02
   12      68   -7.180408e+00   0.000e+00    1.000e+00    1.140e-02   5.584e-03
   13      73   -7.180411e+00   0.000e+00    1.000e+00    1.764e-03   4.677e-04
   14      78   -7.180412e+00   0.000e+00    1.000e+00    8.827e-05   1.304e-05
   15      83   -7.180412e+00   0.000e+00    1.000e+00    1.528e-06   1.023e-07
```

```
Local minimum found that satisfies the constraints.

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

```
Serial FMINCON optimization takes 18.1397 seconds.
```

Minimizing Using Genetic Algorithm

Since `ga` usually takes many more function evaluations than `fmincon`, we remove the expensive constraint from this problem and perform unconstrained optimization

instead; we pass empty ([]) for constraints. In addition, we limit the maximum number of generations to 15 for `ga` so that `ga` can terminate in a reasonable amount of time. We are interested in measuring the time taken by `ga` so that we can compare it to the parallel `ga` evaluation. Note that running `ga` requires Global Optimization Toolbox.

```
rng default % for reproducibility
try
    gaAvailable = false;
    nvar = 4;
    gaoptions = gaoptimset('Generations',15,'Display','iter');
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_sequential = toc(startTime);
    fprintf('Serial GA optimization takes %g seconds.\n',time_ga_sequential);
    gaAvailable = true;
catch ME
    warning(message('optimdemos:optimparfor:gaNotFound'));
end
```

```
                                Best           Mean         Stall
Generation      f-count         f(x)           f(x)      Generations
    1            100        -6.433e+16     -1.287e+15        0
    2            150        -1.501e+17     -7.138e+15        0
    3            200        -7.878e+26     -1.576e+25        0
    4            250        -8.664e+27     -1.466e+26        0
    5            300        -1.096e+28     -2.062e+26        0
    6            350        -5.422e+33     -1.145e+32        0
    7            400        -1.636e+36     -3.316e+34        0
    8            450        -2.933e+36     -1.513e+35        0
    9            500        -1.351e+40     -2.705e+38        0
   10            550        -1.351e+40       -7.9e+38        1
   11            600         -2.07e+40     -2.266e+39        0
   12            650        -1.845e+44     -3.696e+42        0
   13            700        -2.893e+44     -1.687e+43        0
   14            750        -5.076e+44     -6.516e+43        0
   15            800        -8.321e+44     -2.225e+44        0
Optimization terminated: maximum number of generations exceeded.
Serial GA optimization takes 87.3686 seconds.
```

Setting Parallel Computing Toolbox

The finite differencing used by the functions in Optimization Toolbox to approximate derivatives is done in parallel using the `parfor` feature if Parallel Computing Toolbox

is available and there is a parallel pool of workers. Similarly, `ga`, `gamultiobj`, and `patternsearch` solvers in Global Optimization Toolbox evaluate functions in parallel. To use the `parfor` feature, we use the `parpool` function to set up the parallel environment. The computer on which this example is published has four cores, so `parpool` starts four MATLAB® workers. If there is already a parallel pool when you run this example, we use that pool; see the documentation for `parpool` for more information.

```
if max(size(gcp)) == 0 % parallel pool needed
    parpool % create the parallel pool
end
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

Minimizing Using Parallel `fmincon`

To minimize our expensive optimization problem using the parallel `fmincon` function, we need to explicitly indicate that our objective and constraint functions can be evaluated in parallel and that we want `fmincon` to use its parallel functionality wherever possible. Currently, finite differencing can be done in parallel. We are interested in measuring the time taken by `fmincon` so that we can compare it to the serial `fmincon` run.

```
options = optimoptions(options,'UseParallel',true);
startTime = tic;
xsol = fmincon(@expensive_objfun,startPoint,[],[],[],[],[],[],@expensive_confun,options
time_fmincon_parallel = toc(startTime);
fprintf('Parallel FMINCON optimization takes %g seconds.\n',time_fmincon_parallel);
```

					Norm of	First-order
Iter	F-count	f(x)	Feasibility	Steplength	step	optimality
0	5	1.839397e+00	1.500e+00			3.311e+00
1	12	-8.841073e-01	4.019e+00	4.900e-01	2.335e+00	7.015e-01
2	17	-1.382832e+00	0.000e+00	1.000e+00	1.142e+00	9.272e-01
3	22	-2.241952e+00	0.000e+00	1.000e+00	2.447e+00	1.481e+00
4	27	-3.145762e+00	0.000e+00	1.000e+00	1.756e+00	5.464e+00
5	32	-5.277523e+00	6.413e+00	1.000e+00	2.224e+00	1.357e+00
6	37	-6.310709e+00	0.000e+00	1.000e+00	1.099e+00	1.309e+00
7	43	-6.447956e+00	0.000e+00	7.000e-01	2.191e+00	3.631e+00
8	48	-7.135133e+00	0.000e+00	1.000e+00	3.719e-01	1.205e-01
9	53	-7.162732e+00	0.000e+00	1.000e+00	4.083e-01	2.935e-01
10	58	-7.178390e+00	0.000e+00	1.000e+00	1.591e-01	3.110e-02
11	63	-7.180399e+00	1.191e-05	1.000e+00	2.644e-02	1.553e-02
12	68	-7.180408e+00	0.000e+00	1.000e+00	1.140e-02	5.584e-03
13	73	-7.180411e+00	0.000e+00	1.000e+00	1.764e-03	4.677e-04
14	78	-7.180412e+00	0.000e+00	1.000e+00	8.827e-05	1.304e-05
15	83	-7.180412e+00	0.000e+00	1.000e+00	1.528e-06	1.023e-07

```
Local minimum found that satisfies the constraints.

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

```
Parallel FMINCON optimization takes 8.78988 seconds.
```

Minimizing Using Parallel Genetic Algorithm

To minimize our expensive optimization problem using the ga function, we need to explicitly indicate that our objective function can be evaluated in parallel and that we want ga to use its parallel functionality wherever possible. To use the parallel ga we also require that the 'Vectorized' option be set to the default (i.e., 'off'). We are again interested in measuring the time taken by ga so that we can compare it to the serial ga run. Though this run may be different from the serial one because ga uses a random number generator, the number of expensive function evaluations is the same in both runs. Note that running ga requires Global Optimization Toolbox.

```
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = gaoptimset(gaoptions,'UseParallel',true);
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_parallel = toc(startTime);
    fprintf('Parallel GA optimization takes %g seconds.\n',time_ga_parallel);
end
```

Generation	f-count	Best f(x)	Mean f(x)	Stall Generations
1	100	-6.433e+16	-1.287e+15	0
2	150	-1.501e+17	-7.138e+15	0
3	200	-7.878e+26	-1.576e+25	0
4	250	-8.664e+27	-1.466e+26	0
5	300	-1.096e+28	-2.062e+26	0
6	350	-5.422e+33	-1.145e+32	0
7	400	-1.636e+36	-3.316e+34	0
8	450	-2.933e+36	-1.513e+35	0
9	500	-1.351e+40	-2.705e+38	0
10	550	-1.351e+40	-7.9e+38	1

```
11          600        -2.07e+40      -2.266e+39       0
12          650        -1.845e+44     -3.696e+42       0
13          700        -2.893e+44     -1.687e+43       0
14          750        -5.076e+44     -6.516e+43       0
15          800        -8.321e+44     -2.225e+44       0
Optimization terminated: maximum number of generations exceeded.
Parallel GA optimization takes 23.707 seconds.
```

Compare Serial and Parallel Time

```
X = [time_fmincon_sequential time_fmincon_parallel];
Y = [time_ga_sequential time_ga_parallel];
t = [0 1];
plot(t,X,'r--',t,Y,'k-')
ylabel('Time in seconds')
legend('fmincon','ga')
ax = gca;
ax.XTick = [0 1];
ax.XTickLabel = {'Serial' 'Parallel'};
axis([0 1 0 ceil(max([X Y]))])
title('Serial Vs. Parallel Times')
```

Serial Vs. Parallel Times
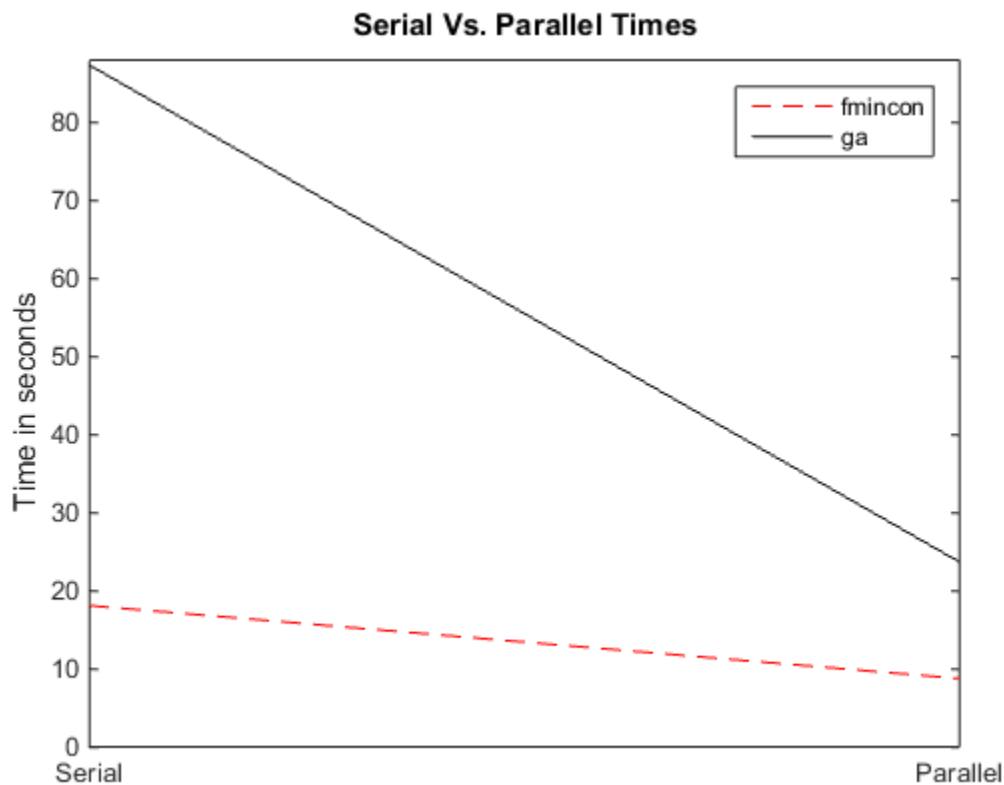
Utilizing parallel function evaluation via `parfor` improved the efficiency of both `fmincon` and `ga`. The improvement is typically better for expensive objective and constraint functions.

At last we delete the parallel pool.

```
if max(size(gcp)) > 0 % parallel pool exists
    delete(gcp) % delete the pool
end
```

```
Parallel pool using the 'local' profile is shutting down.
```

Improving Performance with Parallel Computing

In this section...
"Factors That Affect Speed" on page 12-15
"Factors That Affect Results" on page 12-15
"Searching for Global Optima" on page 12-16

Factors That Affect Speed

Some factors may affect the speed of execution of parallel processing:

- Parallel overhead. There is overhead in calling `parfor` instead of `for`. If function evaluations are fast, this overhead could become appreciable. In particular, solving a problem in parallel can be slower than solving the problem serially.

- No nested `parfor` loops. This is described in "Nested Parallel Functions" on page 12-3. `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.

- When executing serially, `parfor` loops run slower than `for` loops. Therefore, for best performance, ensure that only your outermost parallel loop calls `parfor`. For example, suppose your code calls `fmincon` within a `parfor` loop. For best performance in this case, set the `fmincon` `UseParallel` option to `false`.

- 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 "parfor 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.

If you have a Global Optimization Toolbox license, you can use the MultiStart solver to examine multiple start points in parallel. See "Parallel Computing" and "Parallel MultiStart" in the Global Optimization Toolbox documentation.

13

Argument and Options Reference

Function 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.	fgoalattain, fmincon, fminimax, fseminf, linprog, lsqlin, quadprog
Aeq, beq	The matrix Aeq and vector beq are, respectively, the coefficients of the linear equality constraints and the corresponding right-side vector: Aeq*x = beq.	fgoalattain, fmincon, fminimax, fseminf, linprog, lsqlin, quadprog
C, d	The matrix C and vector d are, respectively, the coefficients of the over or underdetermined linear system and the right-side vector to be solved.	lsqlin, lsqnonneg
f	The vector of coefficients for the linear term in the linear equation f'*x or the quadratic equation x'*H*x+f'*x.	linprog, quadprog
fun	The function to be optimized. fun is either a function handle to a file or is 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
goal	Vector of values that the objectives attempt to attain. The vector is	fgoalattain

Argument	Description	Used by Functions
	the same length as the number of objectives.	
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 2-50 explains how to parameterize the function `nonlcon`, if necessary. See the individual reference pages for more information on `nonlcon`.	`fgoalattain, fmincon, fminimax`
`ntheta`	The number of semi-infinite constraints.	`fseminf`
`options`	A structure that defines options used by the optimization functions. For information about the options, see "Optimization Options Reference" on page 13-7 or the individual function reference pages.	All functions

Argument	Description	Used by Functions
seminfcon	The function that computes the nonlinear inequality and equality constraints *and* the semi-infinite constraints. seminfcon is the name of a function file or MEX-file. "Passing Extra Parameters" on page 2-50 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

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	All functions

Argument	Description	Used by Functions
	reference pages for descriptions of `exitflag` specific to each function, and "Exit Flags and Exit Messages" on page 3-3. 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`.	
`fval`	The value of the objective function `fun` at the solution `x`.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `fzero`, `linprog`, `quadprog`
`grad`	The value of the gradient of `fun` at the solution `x`. 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 the `quasi-newton`, `active-set`, or `sqp` methods, `hessian` is the value of the Quasi-Newton approximation to the Hessian at the solution `x`. See "Hessian" on page 3-26.	`fmincon`, `fminunc`
`jacobian`	The value of the Jacobian of `fun` at the solution `x`. If `fun` does not compute the Jacobian, `jacobian` is a finite-differencing approximation of the Jacobian.	`lsqcurvefit`, `lsqnonlin`, `fsolve`

Argument	Description	Used by Functions
lambda	The Lagrange multipliers at the solution x, see "Lagrange Multiplier Structures" on page 3-25. 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, see "Output Structures" on page 3-24. 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 Reference

In this section...

Optimization Options

The following table describes optimization options. Create options using the `optimoptions` function, or `optimset` for `fminbnd`, `fminsearch`, `fzero`, or `lsqnonneg`.

See the individual function reference pages for information about available 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 `optimoptions(@`*solvername*`)` or the equivalent `optimoptions('`*solvername*`')`. For example,

```
optimoptions(@fmincon)
```

returns a list of the options and the default values for the default `'interior-point'` `fmincon` algorithm. To find the default values for another `fmincon` algorithm, set the `Algorithm` option. For example,

```
opts = optimoptions(@fmincon,'Algorithm','sqp')
```

Optimization Options

Option Name	Description	Used by Functions	Restrictions
Algorithm	Chooses the algorithm used by the solver.	fmincon, fminunc, fsolve, linprog, lsqcurvefit, lsqlin, lsqnonlin, quadprog	
AlwaysHonorConstrai	The default `'bounds'` ensures that bound constraints are	fmincon	

Option Name	Description	Used by Functions	Restrictions
	satisfied at every iteration. Turn off by setting to `'none'`.		
BranchingRule	Rule for choosing the component for branching: • `'maxpscost'` — The fractional component with maximum pseudocost. See "Branch and Bound" on page 8-42. • `'mostfractional'` — The component whose fractional part is closest to `1/2`. • `'maxfun'` — The fractional component with maximal corresponding component in the absolute value of objective vector `f`.	intlinprog	optimoptions only
CutGeneration	Level of cut generation (see "Cut Generation" on page 8-40): • `'none'` — No cuts. Makes `CutGenerationMaxIter` irrelevant. • `'basic'` — Normal cut generation. • `'intermediate'` — Use more cut types. • `'advanced'` — Use most cut types.	intlinprog	optimoptions only
CutGenMaxIter	Number of passes through all cut generation methods before entering the branch-and-bound phase, an integer from 1 through 50. Disable cut generation by setting the	intlinprog	optimoptions only

Option Name	Description	Used by Functions	Restrictions
	`CutGeneration` option to `'none'`.		
`DerivativeCheck`	Compare user-supplied analytic derivatives (gradients or Jacobian, depending on the selected solver) to finite differencing derivatives.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	
`Diagnostics`	Display diagnostic information about the function to be minimized or solved.	All but `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg`	
`DiffMaxChange`	Maximum change in variables for finite differencing.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	
`DiffMinChange`	Minimum change in variables for finite differencing.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	
`Display`	Level of display. • `'off'` displays no output. • `'iter'` displays output at each iteration, and gives the default exit message. • `'iter-detailed'` displays output at each iteration, and gives the technical exit message. • `'notify'` displays output only if the function does not converge, and gives the default exit message.	All. See the individual function reference pages for the values that apply.	

Option Name	Description	Used by Functions	Restrictions
	• `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message. • `'final'` displays just the final output, and gives the default exit message. • `'final-detailed'` displays just the final output, and gives the technical exit message.		
FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are `delta = v.*sign′(x).*max(abs(x),TypicalX)` where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are `delta = v.*max(abs(x),TypicalX);` Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.	`fgoalattain,` `fmincon, fminimax,` `fminunc, fseminf,` `fsolve, lsqcurvefit,` `lsqnonlin`	
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	`fgoalattain,` `fmincon, fminimax,` `fminunc, fseminf,` `fsolve, lsqcurvefit,` `lsqnonlin`	

Option Name	Description	Used by Functions	Restrictions
	their evaluation in fmincon interior-point evaluations if the AlwaysHonorConstraints option is set to 'none'.		
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.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, lsqnonlin	
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.	fgoalattain	
GradConstr	User-defined gradients for the nonlinear constraints.	fgoalattain, fmincon, fminimax	
GradObj	User-defined gradients for the objective functions.	fgoalattain, fmincon, fminimax, fminunc, fseminf	
HessFcn	Function handle to a user-supplied Hessian (see "Hessian" on page 14-60).	fmincon	
Hessian	If 'user-supplied', function uses user-defined Hessian or Hessian information (when	fmincon, fminunc	

Option Name	Description	Used by Functions	Restrictions
	using `HessMult`), for the objective function. If `'off'`, function approximates the Hessian using finite differences.		
HessMult	Handle to a user-supplied Hessian multiply function. For `fmincon`, ignored unless `Hessian` is `'user-supplied'` or `'on'`.	`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.	`fmincon`, `fminunc`	
HessUpdate	Quasi-Newton updating scheme.	`fminunc`	
Heuristics	Algorithm for searching for feasible points (see "Heuristics for Finding Feasible Solutions" on page 8-41): • `'none'` • `'rss'` • `'round'` • `'rins'`	`intlinprog`	optimoptions only
HeuristicsMaxNodes	Strictly positive integer that bounds the number of nodes `intlinprog` can explore in its branch-and-bound search for feasible points. See "Heuristics for Finding Feasible Solutions" on page 8-41.	`intlinprog`	optimoptions only
InitBarrierParam	Initial barrier value.	`fmincon`	
InitDamping	Initial Levenberg-Marquardt parameter.	`fsolve`, `lsqcurvefit`, `lsqnonlin`	optimoptions only

Option Name	Description	Used by Functions	Restrictions
`InitialHessMatrix` This option will be removed in a future release.	Initial quasi-Newton matrix.	`fminunc`	`optimset` only
`InitialHessType` This option will be removed in a future release.	Initial quasi-Newton matrix type.	`fminunc`	`optimset` only
`InitTrustRegionRadi`	Initial radius of the trust region.	`fmincon`	
`IPPreprocess`	Types of integer preprocessing (see "Mixed-Integer Program Preprocessing" on page 8-40): • `'none'` — Use very few integer preprocessing steps. • `'basic'` — Use a moderate number of integer preprocessing steps. • `'advanced'` — Use all available integer preprocessing steps.	`intlinprog`	`optimoptions` only
`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.	`fsolve`, `lsqcurvefit`, `lsqnonlin`	
`JacobMult`	User-defined Jacobian multiply function. Ignored unless `Jacobian` is `'on'` for `fsolve`, `lsqcurvefit`, and `lsqnonlin`.	`fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`	
`JacobPattern`	Sparsity pattern of the Jacobian for finite differencing. The size	`fsolve`, `lsqcurvefit`, `lsqnonlin`	

Option Name	Description	Used by Functions	Restrictions
	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.		
LargeScale Use Algorithm instead	Use large-scale algorithm if possible.	fminunc, fsolve, linprog, lsqcurvefit, lsqlin, lsqnonlin	optimset only
LPMaxIter	Strictly positive integer, the maximum number of simplex algorithm iterations per node during the branch-and-bound process.	intlinprog	optimoptions only
LPPreprocess	Type of preprocessing for the solution to the relaxed linear program (see "Linear Program Preprocessing" on page 8-39): • 'none' — No preprocessing. • 'basic' — Use preprocessing.	intlinprog	optimoptions only
MaxFunEvals	Maximum number of function evaluations allowed.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	
MaxIter	Maximum number of iterations allowed.	All but fzero and lsqnonneg	
MaxNodes	Strictly positive integer that is the maximum number of nodes the solver explores in its branch-and-bound process.	intlinprog	

Option Name	Description	Used by Functions	Restrictions
MaxNumFeasPoints	Strictly positive integer. `intlinprog` stops if it finds `MaxNumFeasPoints` integer feasible points.	`intlinprog`	`optimoptions` only
MaxPCGIter	Maximum number of iterations of preconditioned conjugate gradients method allowed.	`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.	`fmincon`	
MaxSQPIter	Maximum number of iterations of sequential quadratic programming method allowed.	`fgoalattain`, `fmincon`, `fminimax`	
MaxTime	Maximum amount of time in seconds allowed for the algorithm.	`intlinprog`, `linprog`	
MeritFunction	Use goal attainment/minimax merit function (multiobjective) vs. `fmincon` (single objective).	`fgoalattain`, `fminimax`	
MinAbsMax	Number of $F(x)$ to minimize the worst case absolute values.	`fminimax`	
NodeSelection	Choose the node to explore next. • `'simplebestproj'` — Best projection. See "Branch and Bound" on page 8-42. • `'minobj'` — Explore the node with the minimum objective function. • `'mininfeas'` — Explore the node with the minimal sum of integer infeasibilities. See	`intlinprog`	`optimoptions` only

Option Name	Description	Used by Functions	Restrictions
	"Branch and Bound" on page 8-42.		
ObjectiveCutOff	Real greater than -Inf. The default is Inf.	intlinprog	optimoptions only
ObjectiveLimit	If the objective function value goes below ObjectiveLimit and the iterate is feasible, then the iterations halt.	fmincon, fminunc, quadprog	
OutputFcn	Specify one or more user-defined functions that the optimization function calls at each iteration. See "Output Function" on page 13-20 or "intlinprog Output Functions and Plot Functions" on page 13-30.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, intlinprog, lsqcurvefit, lsqnonlin	
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 • @optimplotconstrviolati plots the maximum constraint violation • @optimplotresnorm plots the norm of the residuals • @optimplotfirstorderopt plots the first-order of optimality	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, intlinprog, lsqcurvefit, lsqnonlin. See the individual function reference pages for the values that apply.	

Option Name	Description	Used by Functions	Restrictions
	• `@optimplotstepsize` plots the step size • `@optimplotmilp` plots the gap for mixed-integer linear programs See "Plot Functions" on page 13-29 or "`intlinprog` Output Functions and Plot Functions" on page 13-30.		
`PrecondBandWidth`	Upper bandwidth of preconditioner for PCG. Setting to `'Inf'` uses a direct factorization instead of CG.	`fmincon`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`	
`Preprocess`	Level of LP preprocessing prior to simplex or dual simplex algorithm iterations.	`linprog`	`optimoptions` only
`RelLineSrchBnd`	Relative bound on line search step length.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`	
`RelLineSrchBndDurat`	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`	
`RelObjThreshold`	Nonnegative real. `intlinprog` changes the current feasible solution only when it locates another with an objective function value that is at least `RelObjThreshold` lower: (fold – fnew)/(1 + fold) > RelObjThreshold.	`intlinprog`	`optimoptions` only
`RootLPAlgorithm`	Algorithm for solving linear programs:	`intlinprog`	`optimoptions` only

Option Name	Description	Used by Functions	Restrictions
	• `'dual-simplex'` — Dual simplex algorithm • `'primal-simplex'` — Primal simplex algorithm		
RootLPMaxIter	Nonnegative integer that is the maximum number of simplex algorithm iterations to solve the initial linear programming problem.	`intlinprog`	`optimoptions` only
ScaleProblem	For `fmincon` `interior-point` and `sqp` algorithms, `'obj-and-constr'` causes the algorithm to normalize all constraints and the objective function by their initial values. Disable by setting to the default `'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.	`fmincon`, `fsolve`, `lsqcurvefit`, `lsqnonlin`, `quadprog`	
Simplex Use `Algorithm` instead	If `'on'`, function uses the simplex algorithm.	`linprog`	`optimset` only
SubproblemAlgorithm	Determines how the iteration step is calculated.	`fmincon`	
TolCon	Tolerance on the constraint violation.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `intlinprog`, `linprog`, `lsqlin`, `quadprog`	

Option Name	Description	Used by Functions	Restrictions
`TolConSQP`	Constraint violation tolerance for the inner SQP iteration.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`	
`TolFun`	Termination tolerance on the function value.	`fgoalattain`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `linprog` (interior-point only), `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`	
`TolFunLP`	Nonnegative real where reduced costs must exceed `TolFunLP` for a variable to be taken into the basis.	`intlinprog`	`optimoptions` only
`TolGapAbs`	Nonnegative real. `intlinprog` stops if the difference between the internally calculated upper (U) and lower (L) bounds on the objective function is less than or equal to `TolGapAbs`: $U - L \leq \text{TolGapAbs}$.	`intlinprog`	`optimoptions` only
`TolGapRel`	Real from 0 through 1. `intlinprog` stops if the relative difference between the internally calculated upper (U) and lower (L) bounds on the objective function is less than or equal to `TolGapRel`: $(U - L) / (\text{abs}(U) + 1) \leq$ `TolGapRel`.	`intlinprog`	`optimoptions` only
`TolInteger`	Real from `1e-6` through `1e-3`, where the maximum deviation from integer that a component	`intlinprog`	`optimoptions` only

Option Name	Description	Used by Functions	Restrictions
	of the solution x can have and still be considered an integer. `TolInteger` is not a stopping criterion.		
`TolPCG`	Termination tolerance on the PCG iteration.	`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.	`fmincon`	
`TolProjCGAbs`	Absolute tolerance for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.	`fmincon`	
`TolX`	Termination tolerance on x.	All functions except `linprog` and `lsqlin`, and the `quadprog` `'active-set'` algorithm	
`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. Primarily for scaling finite differences for gradient estimation.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`	
`UseParallel`	When `true`, applicable solvers estimate gradients in parallel. Disable by setting to `false`.	`fgoalattain`, `fmincon`, `fminimax`.	

Output Function

The `Outputfcn` field of `options` 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.

Caution `intlinprog` output functions and plot functions differ from those in other solvers. See "`intlinprog` Output Functions and Plot Functions" on page 13-30.

To set up an output function, do the following:

1   Write the output function as a function file or local function.

2   Use `optimoptions` 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 = optimoptions(@solvername,'OutputFcn', @outfun);
```

specifies `OutputFcn` to be the handle to `outfun`. To specify more than one output function, use the syntax

```
options = optimoptions(@solvername,'OutputFcn',{@outfun, @outfun2});
```

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

See "Output Functions" on page 3-35 for an example of an output function.

"Passing Extra Parameters" on page 2-50 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 13-22 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 13-28 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 13-28 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 "Iterative Display" on page 3-16. 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.

Some `optimValues` fields apply only to specific algorithms:

- AS — `active-set`
- D — `trust-region-dogleg`
- IP — `interior-point`
- LM — `levenberg-marquardt`
- Q — `quasi-newton`
- SQP — `sqp`
- TR — `trust-region`
- TRR — `trust-region-reflective`

Some `optimValues` fields exist in certain solvers or algorithms, but are always filled with empty or zero values, so are meaningless. These fields include:

- `constrviolation` for `fminunc` TR and `fsolve` TRR.
- `procedure` for `fmincon` TRR and SQP, and for `fminunc`.

optimValues Fields

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`attainfactor`	Attainment factor for multiobjective problem. For details, see "Goal Attainment Method" on page 7-3.	`fgoalattain`	None
`cgiterations`	Number of conjugate gradient iterations at current optimization iteration.	`fmincon` (IP, TRR), `fminunc` (TR), `fsolve` (TRR), `lsqcurvefit` (TRR), `lsqnonlin` (TRR)	`CG-iterations` See "Iterative Display" on page 3-16.
`constrviolation`	Maximum constraint violation.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf` `fminunc` TR and `fsolve` TRR provide blank field values.	`Max constraint` or `Feasibility` See "Iterative Display" on page 3-16.
`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.	`fmincon` (TRR), `lsqcurvefit` (TRR), `lsqnonlin` (TRR)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
	See "Degeneracy" on page 13-27.		
`directionalderivative`	Directional derivative in the search direction.	`fgoalattain`, `fmincon` (AS), `fminimax`, `fminunc` (Q), `fseminf`, `fsolve` (LM), `lsqcurvefit` (LM), `lsqnonlin` (LM)	`Directional derivative` See "Iterative Display" on page 3-16.
`firstorderopt`	First-order optimality (depends on algorithm). Final value equals optimization function output `output.firstorderopt`.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	`First-order optimality` See "Iterative Display" on page 3-16.
`funccount`	Cumulative number of function evaluations. Final value equals optimization function output `output.funcCount`.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fsolve`, `fzero`, `fseminf`, `lsqcurvefit`, `lsqnonlin`	`F-count` or `Func-count` See "Iterative Display" on page 3-16.
`fval`	Function value at current point. Final value equals optimization function output `fval`. For `fsolve`, `fval` is the vector function value, and iterative display `f(x)` is the squared norm of this vector.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `fzero`	`f(x)` See "Iterative Display" on page 3-16.
`gradient`	Current gradient of objective function — either analytic gradient	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`,	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
	if you provide it or finite-differencing approximation. Final value equals optimization function output `grad`.	`fsolve`, `lsqcurvefit`, `lsqnonlin`	
`iteration`	Iteration number — starts at `0`. Final value equals optimization function output `output.iterations`.	`fgoalattain`, `fminbnd`,`fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fsolve`, `fseminf`, `fzero`, `lsqcurvefit`, `lsqnonlin`	`Iteration` See "Iterative Display" on page 3-16.
`lambda`	The Levenberg-Marquardt parameter, `lambda`, at the current iteration. See "Levenberg-Marquardt Method" on page 10-7.	`fsolve` (LM), `lsqcurvefit` (LM), `lsqnonlin` (LM)	`Lambda`
`lssteplength`	Actual step length divided by initially predicted step length	`fmincon` (AS, SQP), `fminunc` (Q)	`Steplength` or `Line search steplength` or `Step-size` See "Iterative Display" on page 3-16.
`maxfval`	Maximum function value	`fminimax`	None
`positivedefinite`	`0` if algorithm detects negative curvature while computing Newton step. `1` otherwise.	`fmincon` (TRR), `fminunc` (TR), `fsolve` (TRR), `lsqcurvefit` (TRR), `lsqnonlin` (TRR)	None
`procedure`	Procedure messages.	`fgoalattain`, `fminbnd`, `fmincon`	`Procedure`

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
		(AS), `fminimax`, `fminsearch`, `fseminf`, `fzero` `fmincon` TRR and SQP, and `fminunc` provide blank field values.	See "Iterative Display" on page 3-16.
`ratio`	Ratio of change in the objective function to change in the quadratic approximation.	`fmincon` (TRR), `fminunc` (TR), `fsolve` (TRR), `lsqcurvefit` (TRR), `lsqnonlin` (TRR)	None
`residual`	The residual vector.	`lsqcurvefit`, `lsqnonlin`,	`Residual` See "Iterative Display" on page 3-16.
`resnorm`	2-norm of the residual squared.	`lsqcurvefit`, `lsqnonlin`	`Resnorm` See "Iterative Display" on page 3-16.
`searchdirection`	Search direction.	`fgoalattain`, `fmincon` (AS, SQP), `fminimax`, `fminunc` (Q), `fseminf`, `fsolve` (LM), `lsqcurvefit` (LM), `lsqnonlin` (LM)	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` (D)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
stepsize	Current step size (displacement in x). Final value equals optimization function output output.stepsize.	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	Step-size or Norm of Step See "Iterative Display" on page 3-16.
trustregionradius	Radius of trust region.	fmincon (IP, TRR), fminunc (TR), fsolve (D, TRR), lsqcurvefit (TRR), lsqnonlin (TRR)	Trust-region radius See "Iterative Display" on page 3-16.

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,state)
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,optimValues,state)
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 13-20. For an example of using built-in plot functions, "Using a Plot Function" on page 3-29.

To view a predefined plot function listed for `PlotFcns`, you can open it in the MATLAB Editor. For example, to view the file corresponding to the norm of residuals, enter:

```
edit optimplotresnorm.m
```
You can use any predefined plot function as a template for writing a custom plot function.

`intlinprog` Output Functions and Plot Functions

In this section...
"What Are Output Functions and Plot Functions?" on page 13-30
"Custom Function Syntax" on page 13-31
"`optimValues` Structure" on page 13-31

What Are Output Functions and Plot Functions?

`intlinprog` can call an output function or plot function after certain events occur in the algorithm. These events include completing a phase of the algorithm such as solving the root LP problem, adding cuts, finding a new integer feasible solution, appreciably improving the relative gap, or exploring a number of nodes in a branch-and-bound tree.

Caution `intlinprog` output functions and plot functions differ from those in other solvers. For output functions or plot functions in other Optimization Toolbox solvers, see "Output Function" on page 13-20 and "Plot Functions" on page 13-29.

- There is one built-in output function: `savemilpsolutions`. This function collects all the integer feasible points that the algorithm finds. It puts the feasible points in a matrix named `xIntSol` in your base workspace, where each column is one integer feasible point. It saves the objective function values in a vector named `fIntSol`, where each entry is the objective function of the corresponding column in `xIntSol`.

- There is one built-in plot function: `optimplotmilp`. This function plots the internally-calculated bounds on the best objective function value. For an example of its use, see "Factory, Warehouse, Sales Allocation Model" on page 8-52.

Call output functions or plot functions by passing the `OutputFcn` or `PlotFcns` name-value pairs, including the handle to the output function or plot function. For example,

```
options = optimoptions(@intlinprog,'OutputFcn',@savemilpsolutions,'PlotFcns',@optimplot
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,options);
```

If you have several output functions or plot functions, pass them as a cell array.

```
options = optimoptions(@intlinprog,'OutputFcn',{@savemilpsolutions,@customFcn});
```

Custom Function Syntax

Write your own output function or plot function using this syntax:

```
function stop = customFcn(x,optimValues,state)
```

intlinprog passes the data x, optimValues, and state to your function.

- stop — Set to true to halt intlinprog. Set to false to allow intlinprog to continue.
- x — Either empty [] or an N-by-1 vector that is a feasible point. x is nonempty only when intlinprog finds a new integer feasible solution. x can be nonempty when phase is 'heuristics' or 'branching'.
- optimValues — A structure whose details are in "optimValues Structure" on page 13-31.
- state — A string containing one of these values:
    - 'init' — intlinprog is starting. Use this state to set up any plots or data structures that you need.
    - 'iter' — intlinprog is solving the problem. Access data related to the solver's progress. For example, plot or perform file operations.
    - 'done' — intlinprog has finished solving the problem. Close any files, finish annotating plots, etc.

For examples of writing output or plot functions, see the built-in functions savemilpsolutions.m or optimplotmilp.m.

optimValues Structure

optimValues Field	Meaning
phase	Phase of the algorithm. Possible values: • 'rootlp' — intlinprog solved the root LP problem. • 'cutgen' — intlinprog added cuts and improved the lower bound. • 'heuristics' — intlinprog found new feasible points using heuristics. • 'branching' — intlinprog is creating and exploring nodes in a branch-and-bound tree.

`optimValues` Field	Meaning
`fval`	Best objective function found so far at an integer feasible point. When `phase` = `'rootlp'`, `fval` is the objective function value at the root node, which is not necessarily an integer feasible point.
`lowerbound`	Global lower bound of the objective function value. Empty when `phase` = `'rootlp'`.
`relativegap`	Relative gap between `lowerbound` and `fval`. Empty when `phase` = `'rootlp'` or `numfeaspoints` = `0`.
`numnodes`	Number of explored nodes. Nonzero only when `phase` = `'branching'`.
`numfeaspoints`	Number of integer feasible solutions found.
`time`	Time in seconds spent so far, measured with `tic` and `toc` from the time when `state` = `'init'`.

Functions — Alphabetical List

color

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

Solve multiobjective goal attainment problems

Equation

Finds the minimum of a problem specified by

$$\underset{x,\gamma}{\text{minimize}}\,\gamma \quad \text{such that} \quad \begin{cases} F(x) - weight \cdot \gamma \le goal \\ c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub. \end{cases}$$

weight, *goal*, *b*, and *beq* 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.

x, *lb*, and *ub* can be passed as vectors or matrices; see "Matrix Arguments" on page 2-30.

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 2-50 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`.

Note: See "Iterations Can Violate Constraints" on page 2-32.

`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 `options`. Use `optimoptions` to set these options.

`x = fgoalattain(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 14-5.

Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[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 13-2 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 a function file:

```
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 `optimoptions` 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 = optimoptions('fgoalattain','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 `j`th column of `G` is the gradient of the `j`th 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 = optimoptions('fgoalattain','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 2-35 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 2-50 explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

options   "Options" on page 14-10 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 created with `optimoptions`

Output Arguments

"Function Arguments" on page 13-2 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. `attainfactor` contains the value of γ at the solution. If `attainfactor` is negative, the goals have been overachieved; if `attainfactor` is positive, the goals have been underachieved.
`exitflag`	Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

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.MaxFunEvals`
-1	Stopped by an output function or plot 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 final 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 constraint functions
	`message`	Exit message

Options

Optimization options used by `fgoalattain`. Use `optimoptions` to set or change `options`. See "Optimization Options Reference" on page 13-7 for detailed information.

`Diagnostics`	Display diagnostic information about the function to be minimized or solved. The choices are `'on'` or the default, `'off'`.
`DiffMaxChange`	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
`DiffMinChange`	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
`Display`	Level of display (see "Iterative Display" on page 3-16):

- `'off'` or `'none'` displays no output.

- `'iter'` displays output at each iteration, and gives the default exit message.

- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.

- `'notify'` displays output only if the function does not converge, and gives the default exit message.

- `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message.

- `'final'` (default) displays just the final output, and gives the default exit message.

- `'final-detailed'` displays just the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are `delta = v.*sign´(x).*max(abs(x),TypicalX);` where `sign´(x) = sign(x)` except `sign´(0) = 1`. Central finite differences are `delta = v.*max(abs(x),TypicalX);` Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
FinDiffType	Finite differences, used to estimate gradients, are either `'forward'` (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 could 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. The default, `'off'`, displays no error.
GoalsExactAchieve	Specifies the number of objectives for which it is required for the objective fun to equal the goal goal (a nonnegative integer). Such objectives should be partitioned into the first few elements of F. The default is 0.
GradConstr	Gradient for nonlinear constraint functions defined by the user. When set to `'on'`, fgoalattain expects the constraint function to have four outputs, as described in nonlcon in the "Input Arguments" on page 14-5 section. When set to the default, `'off'`, gradients of the nonlinear constraints are estimated by finite differences.
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. Set to `'on'` to have fgoalattain use a user-defined gradient of the objective function. The default, `'off'`, causes fgoalattain to estimate gradients using finite differences.
MaxFunEvals	Maximum number of function evaluations allowed (a positive integer). The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed (a positive integer). The default is 400. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxSQPIter	Maximum number of SQP iterations allowed (a positive integer). The default is `10*max(numberOfVariables, numberOfInequalities + numberOfBounds)`

MeritFunction	Use goal attainment/minimax merit function if set to `'multiobj'`, the default. Use `fmincon` merit function if set to `'singleobj'`.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`).

- @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

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

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. The default is none (`[]`).
RelLineSrchBndDuration	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is `1`).
TolCon	Termination tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.

TolConSQP	Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is `1e-6`.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on `x`, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-61.
TypicalX	Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fgoalattain` uses `TypicalX` for scaling finite differences for gradient estimation.
UseParallel	When `true`, estimate gradients in parallel. Disable by setting to the default, `false`. See "Parallel Computing".

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 a function 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 = optimoptions('fgoalattain','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 script `goaldemo`. (From the MATLAB Help browser or the MathWorks Web site documentation, you can click the `goaldemo` name to display the example.) After about 11 iterations, a solution is

```
Active inequalities (to within options.TolCon = 1e-006):
  lower       upper       ineqlin    ineqnonlin
    1                                     1
    2                                     2
    4


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 `optimoptions` 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 = optimoptions('fgoalattain','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,fval,attainfactor

K =
    -1.5954     1.2040
    -0.4201    -2.9046

fval =
    -5.0000
    -3.0000
    -1.0000

attainfactor =
  1.1304e-022
```

In this case the optimizer has tried to match the objectives to the goals. The attainment factor (of `1.1304e-22` or so, depending on your system) 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.

Limitations

The objectives must be continuous. `fgoalattain` might give only local solutions.

More About

Algorithms

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 Algorithms" on page 7-2.

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 6-28. 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 7-3 in "Introduction to Algorithms." Setting the `MeritFunction` option to `'singleobj'` with

```
options = optimoptions(options,'MeritFunction','singleobj')
```

uses the merit function and Hessian used in `fmincon`.

See also "SQP Implementation" on page 6-29 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'`.

- "Create Function Handle"
- "Multiobjective Optimization"

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

fmincon | fminimax | optimoptions | optimtool

fminbnd

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 2-50 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 either a function handle to a file or is 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 14-21.

Create the structure `problem` by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[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 13-2 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 a file:

                 `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 14-22 provides the function-specific details for the `options` values.

problem	objective	Objective function
	x1	Left endpoint
	x2	Right endpoint
	solver	`'fminbnd'`
	options	Options structure created using `optimset`

Output Arguments

"Function Arguments" on page 13-2 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.MaxFunEvals`.
	-1	Stopped by an output function or plot function.
	-2	The bounds are inconsistent, meaning `x1 > x2`.
output	Structure containing information about the optimization. The fields of the structure are	
	iterations	Number of iterations taken
	funcCount	Number of function evaluations
	algorithm	`'golden section search, parabolic interpolation'`
	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 Reference" on page 13-7 for detailed information.

Display	Level of display. `'off'` or `'none'` 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. For iterative display details, see "Iterative Display" on page 3-16.
FunValCheck	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. The default `'off'` displays no error.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is `500`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default is `500`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfunccount` plots the function count • `@optimplotfval` plots the function value For information on writing a custom plot function, see "Plot Functions" on page 13-29.
TolX	Termination tolerance on `x`, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-61.

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 a function 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 = myfun(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 function file:

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbnd`. 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)

x =
    0.9999
```

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.

More About

Algorithms

`fminbnd` is a function 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 a point `x` in the interior of the interval (x_1,x_2) that is close to the minimizer. In this case, the distance of `x` from the minimizer is no more than `2*(TolX + 3*abs(x)*sqrt(eps))`. See [1] or [2] for details about the algorithm.

- "Create Function Handle"
- "Anonymous Functions"

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

`fminsearch` | `fmincon` | `fminunc` | `optimset` | `optimtool`

fmincon

Find minimum of constrained nonlinear multivariable function

Nonlinear programming solver.

Finds the minimum of a problem specified by

$$\min_{x} f(x) \text{ such that } \begin{cases} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub, \end{cases}$$

b and beq 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.

x, lb, and ub can be passed as vectors or matrices; see "Matrix Arguments" on page 2-30.

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,output] = fmincon( ___ )
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon( ___ )
```

Description

`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.

Note: "Passing Extra Parameters" on page 2-50 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`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`.

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. See "Iterations Can Violate Constraints" on page 2-32.

`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 `options`. Use `optimoptions` 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 14-43. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[x,fval] = fmincon( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fmincon( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fmincon`, and a structure `output` with information about the optimization process.

```
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon( ___ )
```
additionally returns:

- lambda — Structure with fields containing the Lagrange multipliers at the solution x.

- grad — Gradient of fun at the solution x.

- hessian — Hessian of fun at the solution x. See "fmincon Hessian" on page 3-27.

Examples

Linear Inequality Constraint

Find the minimum value of Rosenbrock's function when there is a linear inequality constraint.

Set the objective function fun to be Rosenbrock's function. Rosenbrock's function is well-known to be difficult to minimize. It has its minimum objective value of 0 at the point (1,1). For more information, see "Solve a Constrained Nonlinear Problem".

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

Find the minimum value starting from the point [-1,2], constrained to have $x(1) + 2x(2) \leq 1$. Express this constraint in the form Ax <= b by taking A = [1,2] and b = 1. Notice that this constraint means that the solution will not be at the unconstrained solution (1,1), because at that point $x(1) + 2x(2) = 3 > 1$.

```
x0 = [-1,2];
A = [1,2];
b = 1;
x = fmincon(fun,x0,A,b)
```

```
Local minimum found that satisfies the constraints.

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

```
x =

    0.5022    0.2489
```

Linear Inequality and Equality Constraint

Find the minimum value of Rosenbrock's function when there are both a linear inequality constraint and a linear equality constraint.

Set the objective function `fun` to be Rosenbrock's function.

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

Find the minimum value starting from the point `[0.5,0]`, constrained to have $x(1) + 2x(2) \leq 1$ and $2x(1) + x(2) = 1$.

- Express the linear inequality constraint in the form `A*x <= b` by taking `A = [1,2]` and `b = 1`.
- Express the linear equality constraint in the form `Aeq*x <= beq` by taking `Aeq = [2,1]` and `beq = 1`.

```
x0 = [0.5,0];
A = [1,2];
b = 1;
Aeq = [2,1];
beq = 1;
x = fmincon(fun,x0,A,b,Aeq,beq)
```

```
Local minimum found that satisfies the constraints.

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

```
x =
```

```
    0.4149    0.1701
```

Bound Constraints

Find the minimum of an objective function in the presence of bound constraints.

The objective function is a simple algebraic function of two variables.

```
fun = @(x)1+x(1)/(1+x(2)) - 3*x(1)*x(2) + x(2)*(1+x(1));
```

Look in the region where x has positive values, $x(1) \le 1$, and $x(2) \le 2$.

```
lb = [0,0];
ub = [1,2];
```

There are no linear constraints, so set those arguments to [].

```
A = [];
b = [];
Aeq = [];
beq = [];
```

Try an initial point in the middle of the region. Find the minimum of fun, subject to the bound constraints.

```
x0 = [0.5,1];
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

```
Local minimum found that satisfies the constraints.

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

<stopping criteria details>


x =

    1.0000    2.0000
```

A different initial point can lead to a different solution.

```
x0 = x0/5;
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

```
Local minimum found that satisfies the constraints.

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

<stopping criteria details>
```

```
x =

   1.0e-06 *

    0.4000    0.4000
```

To see which solution is better, see "Obtain the Objective Function Value" on page 14-38.

Nonlinear Constraints

Find the minimum of a function subject to nonlinear constraints

Find the point where Rosenbrock's function is minimized within a circle, also subject to bound constraints.

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

Look within the region $0 \le x(1) \le 0.5$, $0.2 \le x(2) \le 0.8$.

```
lb = [0,0.2];
ub = [0.5,0.8];
```

Also look within the circle centered at [1/3,1/3] with radius 1/3. Copy the following code to a file on your MATLAB® path named `circlecon.m`.

```
% Copyright 2015 The MathWorks, Inc.

function [c,ceq] = circlecon(x)
```

```
c = (x(1)-1/3)^2 + (x(2)-1/3)^2 - (1/3)^2;
ceq = [];
```

There are no linear constraints, so set those arguments to `[]`.

```
A = [];
b = [];
Aeq = [];
beq = [];
```

Choose an initial point satisfying all the constraints.

```
x0 = [1/4,1/4];
```

Solve the problem.

```
nonlcon = @circlecon;
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

```
Local minimum found that satisfies the constraints.

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

```
x =

    0.5000    0.2500
```

Nondefault Options

Set options to view iterations as they occur and to use a different algorithm.

To observe the `fmincon` solution process, set the `Display` option to `'iter'`. Also, try the `'sqp'` algorithm, which is sometimes faster or more accurate than the default `'interior-point'` algorithm.

```
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
```

Find the minimum of Rosenbrock's function on the unit disk, $\|x\|^2 \leq 1$. First create a function that represents the nonlinear constraint. Save this as a file named `unitdisk.m` on your MATLAB® path.

```
% Copyright 2015 The MathWorks, Inc.

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

Create the remaining problem specifications. Then run `fmincon`.

```
fun = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = @unitdisk;
x0 = [0,0];
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

| | | | | | Norm of | First-order |
Iter	F-count	f(x)	Feasibility	Steplength	step	optimality
0	3	1.000000e+00	0.000e+00			2.000e+00
1	12	8.913011e-01	0.000e+00	1.176e-01	2.353e-01	1.107e+01
2	22	8.047847e-01	0.000e+00	8.235e-02	1.900e-01	1.330e+01
3	28	4.197517e-01	0.000e+00	3.430e-01	1.217e-01	6.153e+00
4	31	2.733703e-01	0.000e+00	1.000e+00	5.254e-02	4.587e-01
5	34	2.397111e-01	0.000e+00	1.000e+00	7.498e-02	3.029e+00
6	37	2.036002e-01	0.000e+00	1.000e+00	5.960e-02	3.019e+00
7	40	1.164353e-01	0.000e+00	1.000e+00	1.459e-01	1.058e+00
8	43	1.161753e-01	0.000e+00	1.000e+00	1.754e-01	7.383e+00
9	46	5.901601e-02	0.000e+00	1.000e+00	1.547e-02	7.278e-01
10	49	4.533081e-02	2.898e-03	1.000e+00	5.393e-02	1.252e-01
11	52	4.567454e-02	2.225e-06	1.000e+00	1.492e-03	1.679e-03
12	55	4.567481e-02	4.406e-12	1.000e+00	2.095e-06	1.501e-05
13	58	4.567481e-02	0.000e+00	1.000e+00	2.160e-09	1.511e-05

```
Local minimum possible. Constraints satisfied.
```

```
fmincon stopped because the size of the current step is less than
the default value of the step size tolerance and constraints are
satisfied to within the default value of the constraint tolerance.
```

```
x =

    0.7864    0.6177
```

Include Gradient

Include gradient evaluation in the objective function for faster or more reliable computations.

Include the gradient evaluation as a conditionalized output in the objective function file. For details, see "Including Derivatives". The objective function is Rosenbrock's function,

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

which has gradient

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

```
% Copyright 2015 The MathWorks, Inc.

function [f,g] = rosenbrockwithgrad(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)];
end
```

Save this code as a file named `rosenbrockwithgrad.m` on your MATLAB® path.

Create options to use the objective function gradient.

```
options = optimoptions('fmincon','GradObj','on');
```

Create the other inputs for the problem. Then call `fmincon`.

```
fun = @rosenbrockwithgrad;
x0 = [-1,2];
A = [];
b = [];
Aeq = [];
beq = [];
lb = [-2,-2];
ub = [2,2];
nonlcon = [];
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
Local minimum found that satisfies the constraints.

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

```
x =

    1.0000    1.0000
```

Use a Problem Structure

Solve the same problem as in "Nondefault Options" on page 14-33 using a problem structure instead of separate arguments.

Create the options and a problem structure. See problem for the field names and required fields.

```
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
```

```
problem.options = options;
problem.solver = 'fmincon';
problem.objective = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
problem.x0 = [0,0];
```

Create a function file for the nonlinear constraint function representing $\mathrm{norm}(x)^2 \le 1$.

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

Save this as a file named `unitdisk.m` on your MATLAB path.

Include the nonlinear constraint function in `problem`.

```
problem.nonlcon = @unitdisk;
```

Solve the problem.

```
x = fmincon(problem)
```

```
                                              Norm of First-order
 Iter F-count            f(x) Feasibility  Steplength      step  optimality
    0       3  1.000000e+00  0.000e+00                            2.000e+00
    1      12  8.913011e-01  0.000e+00   1.176e-01  2.353e-01   1.107e+01
    2      22  8.047847e-01  0.000e+00   8.235e-02  1.900e-01   1.330e+01
    3      28  4.197517e-01  0.000e+00   3.430e-01  1.217e-01   6.153e+00
    4      31  2.733703e-01  0.000e+00   1.000e+00  5.254e-02   4.587e-01
    5      34  2.397111e-01  0.000e+00   1.000e+00  7.498e-02   3.029e+00
    6      37  2.036002e-01  0.000e+00   1.000e+00  5.960e-02   3.019e+00
    7      40  1.164353e-01  0.000e+00   1.000e+00  1.459e-01   1.058e+00
    8      43  1.161753e-01  0.000e+00   1.000e+00  1.754e-01   7.383e+00
    9      46  5.901601e-02  0.000e+00   1.000e+00  1.547e-02   7.278e-01
   10      49  4.533081e-02  2.898e-03   1.000e+00  5.393e-02   1.252e-01
   11      52  4.567454e-02  2.225e-06   1.000e+00  1.492e-03   1.679e-03
   12      55  4.567481e-02  4.406e-12   1.000e+00  2.095e-06   1.501e-05
   13      58  4.567481e-02  0.000e+00   1.000e+00  2.160e-09   1.511e-05
```

```
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the default value of the step size tolerance and constraints are
satisfied to within the default value of the constraint tolerance.
```

```
<stopping criteria details>


x =

    0.7864    0.6177
```

The iterative display and solution are the same as in "Nondefault Options" on page 14-33.

Obtain the Objective Function Value

Call `fmincon` with the `fval` output to obtain the value of the objective function at the solution.

The "Bound Constraints" on page 14-31 example shows two solutions. Which is better? Run the example requesting the `fval` output as well as the solution.

```
fun = @(x)1+x(1)./(1+x(2)) - 3*x(1).*x(2) + x(2).*(1+x(1));
lb = [0,0];
ub = [1,2];
A = [];
b = [];
Aeq = [];
beq = [];
x0 = [0.5,1];
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)

Local minimum found that satisfies the constraints.

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

<stopping criteria details>


x =

    1.0000    2.0000


fval =
```

```
    -0.6667
```

Run the problem using a different starting point `x0`.

```
x0 = x0/5;
[x2,fval2] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

x2 =

   1.0e-06 *

    0.4000    0.4000


fval2 =

    1.0000
```

This solution has an objective function value `fval2 = 1`, which is higher than the first value `fval = -0.6667`. The first solution `x` has a lower local minimum objective function value.

Examine Solution Using Extra Outputs

To easily examine the quality of a solution, request the `exitflag` and `output` outputs.

Set up the problem of minimizing Rosenbrock's function on the unit disk, $\|x\|^2 \leq 1$. First create a function that represents the nonlinear constraint. Save this as a file named `unitdisk.m` on your MATLAB® path.

```
% Copyright 2015 The MathWorks, Inc.
```

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

Create the remaining problem specifications.

```
fun = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
nonlcon = @unitdisk;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
x0 = [0,0];
```

Call fmincon using the fval, exitflag, and output outputs.

```
[x,fval,exitflag,output] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

```
Local minimum found that satisfies the constraints.

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



x =

    0.7864    0.6177


fval =

    0.0457


exitflag =

     1
```

```
output =

          iterations: 24
           funcCount: 84
      constrviolation: 0
            stepsize: 6.9160e-06
           algorithm: 'interior-point'
       firstorderopt: 2.0624e-08
        cgiterations: 4
             message: 'Local minimum found that satisfies the constraints....'
```

- The `exitflag` value 1 indicates that the solution is a local minimum.
- The `output` structure reports several statistics about the solution process.
  In particular, it gives the number of iterations in `output.iterations`,
  number of function evaluations in `output.funcCount`, and the feasibility in
  `output.constrviolation`.

Obtain All Outputs

`fmincon` optionally returns several outputs that you can use for analyzing the reported
solution.

Set up the problem of minimizing Rosenbrock's function on the unit disk. First create a
function that represents the nonlinear constraint. Save this as a file named `unitdisk.m`
on your MATLAB path.

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

Create the remaining problem specifications.

```
fun = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
nonlcon = @unitdisk;
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
x0 = [0,0];
```

Request all `fmincon` outputs.

```
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon

Local minimum found that satisfies the constraints.

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

<stopping criteria details>


x =

    0.7864    0.6177


fval =

    0.0457


exitflag =

     1


output =

          iterations: 24
           funcCount: 84
       constrviolation: 0
            stepsize: 6.9160e-06
           algorithm: 'interior-point'
        firstorderopt: 2.0624e-08
         cgiterations: 4
             message: 'Local minimum found that satisfies the constraints.

Optimizati...'


lambda =

          eqlin: [0x1 double]
```

```
      eqnonlin: [0x1 double]
       ineqlin: [0x1 double]
         lower: [2x1 double]
         upper: [2x1 double]
    ineqnonlin: 0.1215


grad =

   -0.1911
   -0.1501


hessian =

  497.2871 -314.5574
 -314.5574  200.2378
```

- The `lambda.ineqnonlin` output shows that the nonlinear constraint is active at the solution, and gives the value of the associated Lagrange multiplier.
- The `grad` output gives the value of the gradient of the objective function at the solution `x`.
- The `hessian` output is described in "fmincon Hessian" on page 3-27.
- "Constrained Optimization"

Input Arguments

fun — Function to minimize
function handle | function name

Function to minimize, specified as a function handle or function name. `fun` is a function that accepts a vector or array `x` and returns a real scalar `f`, the objective function evaluated at `x`.

Specify `fun` as a function handle for a file:

```
x = fmincon(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
```

```
f = ...              % Compute function value at x
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fmincon(@(x)norm(x)^2,x0,A,b);
```

If you can compute the gradient of `fun` *and* the `GradObj` option is set to `'on'`, as set by

```
options = optimoptions('fmincon','GradObj','on')
```
then `fun` must return the gradient vector `g(x)` in the second output argument.

If you can also compute the Hessian matrix *and* the `Hessian` option is set to `'on'` via `options = optimoptions('fmincon','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. `fun` can give a sparse Hessian. See "Writing Objective Functions" on page 2-17 for details.

If you can also compute the Hessian matrix and the `Algorithm` option is set to `'interior-point'`, there are several ways to pass the Hessian to `fmincon`. For more information, see "Hessian" on page 14-60. For an example using Symbolic Math Toolbox to compute the gradient and Hessian, see "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-80.

The `interior-point` and `trust-region-reflective` algorithms allow you to supply a Hessian multiply function. This function gives the result of a Hessian-times-vector product without computing the Hessian directly. This can save memory. See "Hessian Multiply Function" on page 14-62.

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char | function_handle`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

A — Linear inequality constraints
real matrix

Linear inequality constraints, specified as a real matrix. A is an M-by-N matrix, where M is the number of inequalities, and N is the number of variables (number of elements in x0). For large problems, pass A as a sparse matrix.

A encodes the M linear inequalities
A*x <= b,

where x is the column vector of N variables x(:), and b is a column vector with M elements.

For example, to specify
$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30,$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x-components add up to 1 or less, take A = ones(1,N) and b = 1

Data Types: double

b — Linear inequality constraints
real vector

Linear inequality constraints, specified as a real vector. b is an M-element vector related to the A matrix. If you pass b as a row vector, solvers internally convert b to the column vector b(:). For large problems, pass b as a sparse vector.

b encodes the M linear inequalities
A*x <= b,

where x is the column vector of N variables x(:), and A is a matrix of size M-by-N.

For example, to specify
$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30,$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x-components sum to 1 or less, take `A = ones(1,N)` and `b = 1`

Data Types: `double`

Aeq — Linear equality constraints
real matrix

Linear equality constraints, specified as a real matrix. `Aeq` is an `Me`-by-`N` matrix, where `Me` is the number of equalities, and `N` is the number of variables (number of elements in `x0`). For large problems, pass `Aeq` as a sparse matrix.

`Aeq` encodes the `Me` linear equalities
`Aeq*x = beq`,

where `x` is the column vector of `N` variables `x(:)`, and `beq` is a column vector with `Me` elements.

For example, to specify
$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20$,

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x-components sum to 1, take `Aeq = ones(1,N)` and `beq = 1`

Data Types: `double`

beq — Linear equality constraints
real vector

Linear equality constraints, specified as a real vector. `beq` is an `Me`-element vector related to the `Aeq` matrix. If you pass `beq` as a row vector, solvers internally convert `beq` to the column vector `beq(:)`. For large problems, pass `beq` as a sparse vector.

`beq` encodes the `Me` linear equalities
`Aeq*x = beq`,

where x is the column vector of N variables `x(:)`, and `Aeq` is a matrix of size `Meq`-by-`N`.

For example, to specify
$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20,$

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x-components sum to 1, take `Aeq = ones(1,N)` and `beq = 1`

Data Types: `double`

lb — Lower bounds
real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `lb`, then `lb` specifies that
`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(x0)`, then `lb` specifies that
`x(i) >= lb(i)` for `1 <= i <= numel(lb)`.

In this case, solvers issue a warning.

Example: To specify that all x-components are positive, `lb = zeros(size(x0))`

Data Types: `double`

ub — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `ub`, then `ub` specifies that
`x(i) <= ub(i)` for all `i`.

If `numel(ub) < numel(x0)`, then `ub` specifies that
`x(i) <= ub(i)` for `1 <= i <= numel(ub)`.

In this case, solvers issue a warning.

14-47

Example: To specify that all x-components are less than one, `ub = ones(size(x0))`

Data Types: `double`

`nonlcon` — Nonlinear constraints
function handle | function name

Nonlinear constraints, specified as a function handle or function name. `nonlcon` is a function that accepts a vector or array `x` and returns two arrays, `c(x)` and `ceq(x)`.

*   `c(x)` is the array of nonlinear inequality constraints at `x`. `fmincon` attempts to satisfy
    `c(x) <= 0` for all entries of `c`.
*   `ceq(x)` is the array of nonlinear equality constraints at `x`. `fmincon` attempts to satisfy
    `ceq(x) = 0` for all entries of `ceq`.

For example,

```
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 = optimoptions('fmincon','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)`. `GC` and `GCeq` can be sparse or dense. If `GC` or `GCeq` is large, with relatively few nonzero entries, save running time and memory in the `interior-point` algorithm by representing them as sparse matrices. For more information, see "Nonlinear Constraints" on page 2-35.

Data Types: `char` | `function_handle`

`options` — Optimization options
output of `optimoptions` | structure such as `optimset` returns

Optimization options, specified as the output of `optimoptions` or a structure such as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose the optimization algorithm:
	
	  • `'interior-point'` (default)
	  • `'trust-region-reflective'`
	  • `'sqp'`
	  • `'active-set'`
	
	For information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
	
	The `trust-region-reflective` algorithm 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 you select the `'trust-region-reflective'` algorithm and these conditions are not all satisfied, `fmincon` throws an error.
	
	The `'active-set'` and `'sqp'` algorithms are not large-scale. See "Large-Scale vs. Medium-Scale Algorithms" on page 2-12.

DerivativeCheck	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are `'off'` (default) or `'on'`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. Choices are `'off'` (default) or `'on'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
Display	Level of display (see "Iterative Display" on page 3-16):

- · 'off' or 'none' displays no output.
- · 'iter' displays output at each iteration, and gives the default exit message.
- · 'iter-detailed' displays output at each iteration, and gives the technical exit message.
- · 'notify' displays output only if the function does not converge, and gives the default exit message.
- · 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.
- · 'final' (default) displays only the final output, and gives the default exit message.
- · 'final-detailed' displays only the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set FinDiffRelStep to a vector v, forward finite differences steps delta are
	`delta = v.*sign′(x).*max(abs(x),TypicalX);`
	where sign′(x) = sign(x) except sign′(0) = 1. Central finite differences are
	`delta = v.*max(abs(x),TypicalX);`
	Scalar FinDiffRelStep expands to a vector. The default is sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences.

FinDiffType	Finite differences, used to estimate gradients, are either 'forward' (default), or 'central' (centered). 'central' takes twice as many function evaluations but should be more accurate. The trust-region-reflective algorithm uses FinDiffType only when DerivativeCheck is set to 'on'. fmincon is careful to obey bounds when estimating both types of finite differences. So, for example, it could take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds. However, for the interior-point algorithm, 'central' differences might violate bounds during their evaluation if the AlwaysHonorConstraints option is set to 'none'.

FunValCheck	Check whether objective function values are valid. The default setting, `'off'`, does not perform a check. The `'on'` setting displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`.
GradConstr	Gradient for nonlinear constraint functions defined by the user. When set to the default, `'off'`, fmincon estimates gradients of the nonlinear constraints by finite differences. When set to `'on'`, fmincon expects the constraint function to have four outputs, as described in nonlcon. The `trust-region-reflective` algorithm does not accept nonlinear constraints.
GradObj	Gradient for the objective function defined by the user. See the description of fun to see how to define the gradient in `fun`. The default, `'off'`, causes fmincon to estimate gradients using finite differences. Set to `'on'` to have fmincon use a user-defined gradient of the objective function. You must provide the gradient, and set `GradObj` to `'on'`, to use the trust-region-reflective method.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default value for all algorithms except `interior-point` is `100*numberOfVariables`; for the `interior-point` algorithm the default is `3000`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default value for all algorithms except `interior-point` is `400`; for the `interior-point` algorithm the default is `1000`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20.

PlotFcns

Plot various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`).

- `@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 optimality measure

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

TolCon

Tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.

TolFun

Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.

TolX

Termination tolerance on `x`, a positive scalar. The default value for all algorithms except `'interior-point'` is `1e-6`; for the `'interior-point'` algorithm, the default is `1e-10`. See "Tolerances and Stopping Criteria" on page 2-61.

TypicalX

Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fmincon` uses `TypicalX` for scaling finite differences for gradient estimation.

The `'trust-region-reflective'` algorithm uses `TypicalX` only for the `DerivativeCheck` option.

UseParallel	When `true`, fmincon estimates gradients in parallel. Disable by setting to the default, `false`. `trust-region-reflective` requires a gradient in the objective, so `UseParallel` does not apply. See "Parallel Computing".

Trust-Region-Reflective Algorithm

Hessian	If `'off'` (default), fmincon approximates the Hessian using finite differences. If `'on'` or `'user-supplied'`, fmincon uses a user-defined Hessian (defined in fun), or Hessian information (when using `HessMult`), for the objective function. See "Hessian" on page 14-60.
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 is 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. The matrix `W = H*Y`, although `H` is not formed explicitly. fmincon uses `Hinfo` to compute the preconditioner. For information on how to supply values for any additional parameters hmfun needs, see "Passing Extra Parameters" on page 2-50.

Note Hessian must be set to `'on'` or `'user-supplied'` for fmincon to pass Hinfo from fun to hmfun.

See "Hessian Multiply Function" on page 14-62. See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75 for an example.

HessPattern	Sparsity pattern of the Hessian for finite differencing. Set HessPattern(i,j) = 1 when you can have ∂^2fun/∂x(i)∂x(j) $\neq 0$. Otherwise, set HessPattern(i,j) = 0.
	Use HessPattern when it is inconvenient to compute the Hessian matrix H in fun, but you can determine (say, by inspection) when the ith component of the gradient of fun depends on x(j). fmincon can approximate H via sparse finite differences (of the gradient) if you provide the *sparsity structure* of H as the value for HessPattern. In other words, provide the locations of the nonzeros.
	When the structure is unknown, do not set HessPattern. The default behavior is as if HessPattern is a dense matrix of ones. Then fmincon computes a full finite-difference approximation in each iteration. This computation can be very expensive for large problems, so it is usually better to determine the sparsity structure.
MaxPCGIter	Maximum number of preconditioned conjugate gradient (PCG) iterations, a positive scalar. The default is max(1,floor(numberOfVariables/2)). For more information, see "Preconditioned Conjugate Gradient Method" on page 6-23.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG, a nonnegative integer. 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, a positive scalar. The default is 0.1.
Active-Set Algorithm	
MaxSQPIter	Maximum number of SQP iterations allowed, a positive integer. The default is 10*max(numberOfVariables, numberOfInequalities + numberOfBounds).

RelLineSrchBnd | Relative bound (a real nonnegative scalar value) on the line search step length. 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. The default is no bounds ([ ]).

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, a positive scalar. The default is `1e-6`.

Interior-Point Algorithm

AlwaysHonorConstraints | The default `'bounds'` ensures that bound constraints are satisfied at every iteration. Disable by setting to `'none'`.

HessFcn | Function handle to a user-supplied Hessian (see "Hessian" on page 14-60). This is used when the `Hessian` option is set to `'user-supplied'`.

Hessian | Chooses how `fmincon` calculates the Hessian (see "Hessian" on page 14-60). The choices are:

- `'bfgs'` (default)
- `'fin-diff-grads'`
- `'lbfgs'`
- `{'lbfgs',Positive Integer}`
- `'user-supplied'`

HessMult | Handle to a user-supplied function that gives a Hessian-times-vector product (see "Hessian Multiply Function" on page 14-62). This is used when the `Hessian` option is set to `'user-supplied'`.

InitBarrierParam | Initial barrier value, a positive scalar. 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, a positive scalar. 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. This positive integer has a default value of `2*(numberOfVariables - numberOfEqualities)`.
ObjectiveLimit	A tolerance (stopping criterion) that is a scalar. If the objective function value goes below `ObjectiveLimit` and the iterate is feasible, the iterations halt, because the problem is presumably unbounded. The default value is `-1e20`.
ScaleProblem	`'obj-and-constr'` causes the algorithm to normalize all constraints and the objective function. Disable by setting to the default `'none'`.
SubproblemAlgorithm	Determines how the iteration step is calculated. The default, `'ldl-factorization'`, is usually faster than `'cg'` (conjugate gradient), though `'cg'` might 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. This positive scalar has a default of `0.01`.
TolProjCGAbs	Absolute tolerance (stopping criterion) for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration. This positive scalar has a default of `1e-10`.

SQP Algorithm

ObjectiveLimit	A tolerance (stopping criterion) that is a scalar. If the objective function value goes below `ObjectiveLimit` and the iterate is feasible, the iterations halt, because the problem is presumably unbounded. The default value is `-1e20`.
ScaleProblem	`'obj-and-constr'` causes the algorithm to normalize all constraints and the objective function. Disable by setting to the default `'none'`.

Example: `options = optimoptions('fmincon','GradObj','on','GradConstr','on')`

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
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 created with optimoptions

You must supply at least the objective, x0, solver, and options fields in the problem structure.

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: struct

Output Arguments

x — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when exitflag is positive. For information on the quality of the solution, see "When the Solver Succeeds" on page 4-22.

fval — Objective function value at solution
real number

Objective function value at the solution, returned as a real number. Generally, fval = fun(x).

exitflag — Reason `fmincon` stopped
integer

Reason `fmincon` stopped, returned as an integer.

All Algorithms:

1	First-order optimality measure was less than `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.MaxFunEvals`.
-1	Stopped by an output function or plot function.
-2	No feasible point was found.

`trust-region-reflective`, `interior-point`, and `sqp` algorithms:

2	Change in `x` was less than `options.TolX` and maximum constraint violation was less than `options.TolCon`.

`trust-region-reflective` algorithm only:

3	Change in the objective function value was less than `options.TolFun` and maximum constraint violation was less than `options.TolCon`.

`active-set` algorithm only:

4	Magnitude of the search direction was less than `2*options.TolX` and maximum 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`.

`interior-point` and `sqp` algorithms:

-3	Objective function at current iteration went below `options.ObjectiveLimit` and maximum constraint violation was less than `options.TolCon`.

output — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

iterations	Number of iterations taken
funcCount	Number of function evaluations
lssteplength	Size of line search step relative to search direction (`active-set` and `sqp` algorithms only)
constrviolation	Maximum of constraint functions
stepsize	Length of last displacement in x (not in `active-set` algorithm)
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

lambda — Lagrange multipliers at the solution
structure

Lagrange multipliers at the solution, returned as a structure with fields:

lower	Lower bounds corresponding to lb
upper	Upper bounds corresponding to ub
ineqlin	Linear inequalities corresponding to A and b
eqlin	Linear equalities corresponding to Aeq and beq
ineqnonlin	Nonlinear inequalities corresponding to the c in nonlcon
eqnonlin	Nonlinear equalities corresponding to the ceq in nonlcon

grad — Gradient at the solution
real vector

Gradient at the solution, returned as a real vector. grad gives the gradient of fun at the point x(:).

hessian — Approximate Hessian
real matrix

Approximate Hessian, returned as a real matrix. For the meaning of `hessian`, see "Hessian" on page 3-26.

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.

- For the `'trust-region-reflective'` algorithm, you must provide the gradient in `fun`.

- 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 trust-region-reflective algorithm. Use
  either interior-point or SQP algorithms 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`. See "Hessian" on page 14-60.

  - 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`.

- When the problem is infeasible, `fmincon` attempts to minimize the maximum constraint value.

More About

Hessian

`fmincon` uses a Hessian as an optional input. This Hessian is the matrix of second derivatives of the Lagrangian (see Equation 3-1), namely,

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

The various `fmincon` algorithms handle input Hessians differently:

- The `active-set` and `sqp` algorithms do not accept a user-supplied Hessian. They compute a quasi-Newton approximation to the Hessian of the Lagrangian.

- The `trust-region-reflective` algorithm 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 Scalar Objective Functions" on page 2-18 for details on how to pass the Hessian to `fmincon`. Indicate that you are supplying a Hessian by

  ```
  options = optimoptions('fmincon','Algorithm','trust-region-reflective','Hessian','us
  ```
  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. If `hessian` is large and has relatively few nonzero entries, save running time and memory by representing `hessian` as a sparse matrix. `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 14-1. Indicate that you are supplying a Hessian by

  ```
  options = optimoptions('fmincon','Algorithm','interior-point',...
      'Hessian','user-supplied','HessFcn',@hessianfcn);
  ```

  For an example, see "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-50.

The `interior-point` algorithm has several more options for Hessians; see "Choose Input Hessian for interior-point fmincon" on page 2-21:

- ```
  options = optimoptions('fmincon','Hessian','bfgs');
  ```

  `fmincon` calculates the Hessian by a dense quasi-Newton approximation. This is the default Hessian approximation.

- ```
  options = optimoptions('fmincon','Hessian','lbfgs');
  ```

fmincon calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation. The default memory, 10 iterations, is used.

• `options = optimoptions('fmincon','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 = optimoptions('fmincon','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).

Hessian Multiply Function

The `interior-point` and `trust-region-reflective` algorithms allow you to supply a Hessian multiply function. This function gives the result of a Hessian-times-vector product, without computing the Hessian directly. This can save memory.

The syntaxes for the two algorithms differ:

• For the `interior-point` algorithm, the syntax is

`W = HessMultFcn(x,lambda,v);`

The result `W` should be the product `H*v`, where `H` is the Hessian of the Lagrangian at `x` (see Equation 14-1), `lambda` is the Lagrange multiplier (computed by `fmincon`), and `v` is a vector of size n-by-1. Set options as follows:

`options = optimoptions('fmincon','Algorithm','interior-point','Hessian','user-suppli`
`'SubproblemAlgorithm','cg','HessMult',@HessMultFcn);`

Supply the function `HessMultFcn`, which returns an n-by-1 vector, where n is the number of dimensions of x. The `HessMult` option enables you to pass the result of multiplying the Hessian by a vector without calculating the Hessian.

• The `trust-region-reflective` algorithm does not involve `lambda`:

`W = HessMultFcn(H,v);`

The result `W = H*v`. `fmincon` passes `H` as the value returned in the third output of the objective function (see "Writing Scalar Objective Functions" on page 2-18). `fmincon` also passes `v`, a vector or matrix with *n* rows. The number of columns in `v` can vary, so write `HessMultFcn` to accept an arbitrary number of columns. `H` does not have to be the Hessian; rather, it can be anything that enables you to calculate `W = H*v`.

Set options as follows:

```
options = optimoptions('fmincon','Algorithm','trust-region-reflective',...
    'Hessian','user-supplied','HessMult',@HessMultFcn);
```

For an example using a Hessian multiply function with the `trust-region-reflective` algorithm, see "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75.

Algorithms

Interior-Point Optimization

This algorithm is described in "fmincon Interior Point Algorithm" on page 6-37. There is more extensive description in [1], [41], and [9].

SQP Optimization

The `fmincon` `'sqp'` algorithm is similar to the `'active-set'` algorithm described in "Active-Set Optimization" on page 14-63. "fmincon SQP Algorithm" on page 6-36 describes the main differences. In summary, these differences are:

- "Strict Feasibility With Respect to Bounds" on page 6-36
- "Robustness to Non-Double Results" on page 6-36
- "Refactored Linear Algebra Routines" on page 6-36
- "Reformulated Feasibility Routines" on page 6-36

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 6-26 describes this algorithm in detail.

See also "SQP Implementation" on page 6-29 for more details on the algorithm used.

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 6-21.

- "Optimization Problem Setup"
- "Constrained Nonlinear Optimization Algorithms" on page 6-21

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, 2000, pp. 149–185.

[2] Byrd, R. H., Mary E. Hribar, and Jorge Nocedal. "An Interior Point Algorithm for Large-Scale Nonlinear Programming." *SIAM Journal on Optimization*, Vol 9, No. 4, 1999, pp. 877–900.

[3] Coleman, T. F. and Y. Li. "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds." *SIAM Journal on Optimization*, Vol. 6, 1996, pp. 418–445.

[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, 1994, pp. 189–224.

[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." *Journal of Optimization Theory and Applications*, Vol. 22, 1977, pp. 297.

[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, 2006, pp. 391–408.

See Also

fminbnd | fminsearch | fminunc | optimoptions | optimtool

Introduced before R2006a

fminimax

Solve minimax constraint problem

Equation

Finds the minimum of a problem specified by

$$\min_{x} \max_{i} F_i(x) \text{ such that } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases}$$

where b and beq 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.

x, lb, and ub can be passed as vectors or matrices; see "Matrix Arguments" on page 2-30.

You can also solve max-min problems with `fminimax`, using the identity

$$\max_{x} \min_{i} F_i(x) = -\min_{x} \max_{i} \left(-F_i(x) \right).$$

You can solve problems of the form

$$\min_{x} \max_{i} \left| F_i(x) \right|$$

by using the `MinAbsMax` option; see "Notes" on page 14-77.

Syntax

```
x = fminimax(fun,x0)
x = fminimax(fun,x0,A,b)
x = fminimax(fun,x0,A,b,Aeq,beq)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
x = fminimax(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 2-50 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,x0,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,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.

Note: See "Iterations Can Violate Constraints" on page 2-32.

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 options. Use optimoptions to set these options.

`x = fminimax(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 14-68.

Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[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 13-2 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 a file:

```
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 abs{$F(x)$} }), partition those objectives into the first elements of `F` and use `optimoptions` 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 = optimoptions('fminimax','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)`).

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 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 = optimoptions('fminimax','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 2-35 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`.

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 created with `optimoptions`

Output Arguments

"Function Arguments" on page 13-2 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.MaxFunEvals`.
	-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 constraint functions
message	Exit message

Options

Optimization options used by fminimax. Use optimoptions to set or change options. See "Optimization Options Reference" on page 13-7 for detailed information.

Diagnostics	Display diagnostic information about the function to be minimized or solved. The choices are 'on' or the default, 'off'.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.
Display	Level of display (see "Iterative Display" on page 3-16): • 'off' or 'none' displays no output. • 'iter' displays output at each iteration, and gives the default exit message. • 'iter-detailed' displays output at each iteration, and gives the technical exit message.

	• `'notify'` displays output only if the function does not converge, and gives the default exit message.
	• `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message.
	• `'final'` (default) displays just the final output, and gives the default exit message.
	• `'final-detailed'` displays just the final output, and gives the technical exit message.
`FinDiffRelStep`	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are `delta = v.*sign′(x).*max(abs(x),TypicalX);` where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are `delta = v.*max(abs(x),TypicalX);` Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
`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 could 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`. The default `'off'` displays no error.

GradConstr	Gradient for the user-defined constraints. When set to `'on'`, fminimax expects the constraint function to have four outputs, as described in `nonlcon` in "Input Arguments" on page 14-68. When set to the default `'off'`, fminimax estimates gradients of the nonlinear constraints by finite differences.
GradObj	Gradient for the user-defined objective function. See the preceding description of fun to see how to define the gradient in fun. Set to `'on'` to have fminimax use a user-defined gradient of the objective function. The default `'off'` causes fminimax to estimate gradients using finite differences.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default value is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default value is `400`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxSQPIter	Maximum number of SQP iterations allowed, a positive integer. The default is `10*max(numberOfVariables, numberOfInequalities + numberOfBounds)`.
MeritFunction	Use the goal attainment/minimax merit function if set to `'multiobj'` (default). Use the fmincon merit function if set to `'singleobj'`.
MinAbsMax	Number of elements of $F_i(x)$ to minimize the maximum absolute value of F_i. See "Notes" on page 14-77. The default is `0`.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See "Output Function" on page 13-20.

PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none ([ ]).

- `@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.

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

RelLineSrchBnd	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in x satisfies $|\Delta x(i)| \leq relLineSrchBnd \cdot \max(|x(i)|, |typicalx(i)|)$. This option provides control over the magnitude of the displacements in x for cases in which the solver takes steps that it considers too large. The default is no bounds ([ ]).
RelLineSrchBndDuration	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is 1).
TolCon	Termination tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolConSQP	Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is `1e-6`.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on x, a positive scalar. The default value is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.

TypicalX	Typical x values. The number of elements in TypicalX is equal to the number of elements in x0, the starting point. The default value is ones(numberofvariables,1). fminimax uses TypicalX for scaling finite differences for gradient estimation.
UseParallel	When true, estimate gradients in parallel. Disable by setting to the default false. See "Parallel Computing".

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 a 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,fval

x =
      4.0000
      4.0000
fval =
      0.0000   -64.0000   -2.0000   -8.0000   -0.0000
```

Notes

You can solve problems of the form

$$\min_{x} \max_{i} G_i(x),$$

where

$$G_i(x) = \begin{cases} |F_i(x)| & 1 \le i \le m \\ F_i(x) & i > m. \end{cases}$$

Here m is the value of the `MinAbsMax` option. The advantage of this formulation is you can minimize the absolute value of some components of F, even though the absolute value function is not smooth.

In order to use this option, reorder the elements of F, if necessary, so the first elements are those for which you want the minimum absolute value.

For example, consider the problem in "Examples" on page 14-76. Modify the problem to find the minimum of the maximum absolute values of all $f_i(x)$. Solve this problem by invoking `fminimax` with the commands

```
x0 = [0.1; 0.1]; % Make a starting guess at the solution
options = optimoptions('fminimax','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
```

Limitations

The function to be minimized must be continuous. `fminimax` might only give local solutions.

More About

Algorithms

`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 14-66, 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 `optimoptions` to set the `MeritFunction` option to `'singleobj'` uses the merit function and Hessian used in `fmincon`.

See also "SQP Implementation" on page 6-29 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'`.

- "Create Function Handle"
- "Multiobjective Optimization"

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

fgoalattain | lsqnonlin | optimoptions | optimtool

fminsearch

Find minimum of unconstrained multivariable function using derivative-free method

Equation

Finds the minimum of a problem specified by

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

where $f(x)$ is a function that returns a scalar.

x is a vector or a matrix; see "Matrix Arguments" on page 2-30.

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 2-50 explains how to pass extra parameters to the objective function, if necessary.

---

x = fminsearch(fun,x0) starts at the point x0 and returns a value x that is a local minimizer of the function described in fun. fun is either a function handle to a file or is an anonymous function. x0 can be a scalar, vector, or matrix.

$x = \text{fminsearch(fun,x0,options)}$ 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 14-81.

Create the structure `problem` by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[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 13-2 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 a file:

```
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 14-82 provides the function-specific details for the `options` values.	
problem	objective	Objective function

x0	Initial point for x
solver	`'fminsearch'`
options	Options structure created using `optimset`

Output Arguments

"Function Arguments" on page 13-2 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.MaxFunEvals`.
	-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	`'Nelder-Mead simplex direct search'`
	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 Reference" on page 13-7 for detailed information.

Display	Level of display (see "Iterative Display" on page 3-16):

	• `'off'` or `'none'` 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 or NaN. The default `'off'` displays no error.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is `200*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default value is `200*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`):
	• `@optimplotx` plots the current point.
	• `@optimplotfunccount` plots the function count.
	• `@optimplotfval` plots the function value.
	For information on writing a custom plot function, see "Plot Functions" on page 13-29.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-61. Unlike other solvers, fminsearch stops when it satisfies *both* `TolFun` and `TolX`.

TolX   Termination tolerance on `x`, a positive scalar. The default value is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-61. Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` and `TolX`.

Examples

Example 1: Minimizing Rosenbrock's Function with fminsearch

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
```

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

$$\min_{x} \left\| f(x) \right\|_2^2 = \min_{x} \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$

Instead use the `lsqnonlin` function, which has been optimized for problems of this form.

More About

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`. The algorithm is described in detail in "fminsearch Algorithm" on page 6-10.

`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.

- "Create Function Handle"
- "Anonymous Functions"

References

[1] Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9, Number 1, pp. 112–147, 1998.

See Also

`fminbnd` | `fminunc` | `optimset` | `optimtool`

fminunc

Find minimum of unconstrained multivariable function

Nonlinear programming solver.

Finds the minimum of a problem specified by

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

where $f(x)$ is a function that returns a scalar.

x is a vector or a matrix; see "Matrix Arguments" on page 2-30.

Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(problem)
[x,fval] = fminunc( ___ )
[x,fval,exitflag,output] = fminunc( ___ )
[x,fval,exitflag,output,grad,hessian] = fminunc( ___ )
```

Description

`x = fminunc(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`. The point `x0` can be a scalar, vector, or matrix.

Note: "Passing Extra Parameters" on page 2-50 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

`fminunc` is for nonlinear problems without constraints. If your problem has constraints, generally use `fmincon`. See "Optimization Decision Table" on page 2-5.

`x = fminunc(fun,x0,options)` minimizes `fun` with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fminunc(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 14-93. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[x,fval] = fminunc( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fminunc( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fminunc`, and a structure `output` with information about the optimization process.

`[x,fval,exitflag,output,grad,hessian] = fminunc( ___ )` additionally returns:

- `grad` — Gradient of `fun` at the solution `x`.
- `hessian` — Hessian of `fun` at the solution `x`. See "fminunc Hessian" on page 3-26.

Examples

Minimize a Polynomial

Minimize the function $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2 - 4x_1 + 5x_2$.

Write an anonymous function that calculates the objective.

```
fun = @(x)3*x(1)^2 + 2*x(1)*x(2) + x(2)^2 - 4*x(1) + 5*x(2);
```

Call `fminunc` to find a minimum of `fun` near `[1,1]`.

```
x0 = [1,1];
[x,fval] = fminunc(fun,x0);
```

After a few iterations, `fminunc` returns the solution, `x`, and the value of the function at `x`, `fval`.

```
x,fval

x =

    2.2500   -4.7500
```

```
fval =

  -16.3750
```

Supply the Gradient

`fminunc` can be faster and more reliable when you provide derivatives.

Write an objective function that returns the gradient as well as the function value. Use the conditionalized form described in "Including Derivatives" on page 2-20. The objective function is Rosenbrock's function,

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

which has gradient

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

```matlab
function [f,g] = rosenbrockwithgrad(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)];
end
```

Save this code as a file on your MATLAB path, named `rosenbrockwithgrad.m`.

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```matlab
options = optimoptions('fminunc','Algorithm','trust-region','GradObj','on');
```

Set the initial point to `[-1,2]`. Then call `fminunc`.

```matlab
x0 = [-1,2];
fun = @rosenbrockwithgrad;
x = fminunc(fun,x0,options)
```

```
Local minimum found.
```

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

<stopping criteria details>


x =

    1.0000    1.0000
```

Use a Problem Structure

Solve the same problem as in "Supply the Gradient" on page 14-89 using a problem structure instead of separate arguments.

Write an objective function that returns the gradient as well as the function value. Use the conditionalized form described in "Including Derivatives" on page 2-20. The objective function is Rosenbrock's function,

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

which has gradient

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

```
function [f,g] = rosenbrockwithgrad(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)];
end
```

Save this code as a file on your MATLAB path, named `rosenbrockwithgrad.m`.

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','GradObj','on');
```

Create a problem structure including the initial point `x0 = [-1,2]`.

```
problem.options = options;
problem.x0 = [-1,2];
problem.objective = @rosenbrockwithgrad;
problem.solver = 'fminunc';
```

Solve the problem.

```
x = fminunc(problem)

Local minimum found.

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

<stopping criteria details>


x =

    1.0000    1.0000
```

Obtain the Optimal Objective Function Value

Find both the location of the minimum of a nonlinear function and the value of the function at that minimum.

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2 / 20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Find the location and objective function value of the minimizer starting at `x0 = [1,2]`.

```
x0 = [1,2];
[x,fval] = fminunc(fun,x0)

Local minimum found.

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

<stopping criteria details>
```

```
x =

   -0.6691      0.0000


fval =

   -0.4052
```

Examine the Solution Process

Choose `fminunc` options and outputs to examine the solution process.

Set options to obtain iterative display and use the `'quasi-newton'` algorithm.

```
options = optimoptions(@fminunc,'Display','iter','Algorithm','quasi-newton');
```

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2 / 20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Start the minimization at `x0 = [1,2]`, and obtain outputs that enable you to examine the solution quality and process.

```
x0 = [1,2];
[x,fval,exitflag,output] = fminunc(fun,x0,options)
```

```
                                                First-order
 Iteration  Func-count        f(x)       Step-size     optimality
     0           3        0.256738                        0.173
     1           6        0.222149            1           0.131
     2           9         0.15717            1           0.158
     3          18       -0.227902     0.438133           0.386
     4          21       -0.299271            1           0.46
     5          30       -0.404028     0.102071           0.0458
     6          33       -0.404868            1           0.0296
     7          36       -0.405236            1           0.00119
     8          39       -0.405237            1           0.000252
     9          42       -0.405237            1           7.97e-07

Local minimum found.

Optimization completed because the size of the gradient is less than
```

the default value of the function tolerance.

```
x =

   -0.6691    0.0000


fval =

   -0.4052


exitflag =

     1


output =

        iterations: 9
         funcCount: 42
          stepsize: 2.9343e-04
       lssteplength: 1
      firstorderopt: 7.9721e-07
         algorithm: 'quasi-newton'
           message: 'Local minimum found....'
```

- The exit flag 1 shows that the solution is a local optimum.
- The output structure shows the number of iterations, number of function evaluations, and other information.
- The iterative display also shows the number of iterations and function evaluations.

- "Unconstrained Optimization"

Input Arguments

fun — Function to minimize
function handle | function name

Function to minimize, specified as a function handle or function name. `fun` is a function that accepts a vector or array `x` and returns a real scalar `f`, the objective function evaluated at `x`.

Specify `fun` as a function handle for a file:

```
x = fmincon(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...              % Compute function value at x
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fmincon(@(x)norm(x)^2,x0,A,b);
```

If you can compute the gradient of `fun` *and* the `GradObj` option is set to `'on'`, as set by

```
options = optimoptions('fminunc','GradObj','on')
```
then `fun` must return the gradient vector $g(x)$ in the second output argument.

If you can also compute the Hessian matrix *and* the `Hessian` option is set to `'on'` via `options = optimoptions('fminunc','Hessian','user-supplied')` *and* the `Algorithm` option is set to `'trust-region-reflective'`, `fun` must return the Hessian value $H(x)$, a symmetric matrix, in a third output argument. `fun` can give a sparse Hessian. See "Writing Objective Functions" on page 2-17 for details.

The `trust-region-reflective` algorithm allows you to supply a Hessian multiply function. This function gives the result of a Hessian-times-vector product without computing the Hessian directly. This can save memory. See "Hessian Multiply Function" on page 14-62.

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char | function_handle`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

options — Optimization options
output of `optimoptions` | structure such as `optimset` returns

Optimization options, specified as the output of `optimoptions` or a structure such as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose the `fminunc` algorithm. Choices are `'trust-region'` (default) or `'quasi-newton'`.
If you use `optimset` (not recommended, see "Choose Between optimoptions and optimset" on page 2-57), use `LargeScale` instead of `Algorithm`.	The `'trust-region'` algorithm requires you to provide the gradient (see the description of fun), or else `fminunc` uses the `'quasi-newton'` algorithm. For information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
DerivativeCheck	Compare user-supplied derivatives (gradient of objective) to finite-differencing derivatives. Choices are `'off'` (default) or `'on'`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. Choices are `'off'` (default) or `'on'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
Display	Level of display (see "Iterative Display" on page 3-16):

Display options (continued):

- `'off'` or `'none'` displays no output.
- `'iter'` displays output at each iteration, and gives the default exit message.
- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.
- `'notify'` displays output only if the function does not converge, and gives the default exit message.

- `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message.

- `'final'` (default) displays only the final output, and gives the default exit message.

- `'final-detailed'` displays only the final output, and gives the technical exit message.

FinDiffRelStep
Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are
`delta = v.*sign′(x).*max(abs(x),TypicalX);`
where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are
`delta = v.*max(abs(x),TypicalX);`
Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.

The trust-region algorithm uses `FinDiffRelStep` only when `DerivativeCheck` is set to `'on'`.

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 trust-region algorithm uses `FinDiffType` only when `DerivativeCheck` is set to `'on'`.

FunValCheck
Check whether objective function values are valid. The default setting, `'off'`, does not perform a check. The `'on'` setting displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`.

GradObj
Gradient for the objective function defined by the user. See the description of fun to see how to define the gradient in `fun`. Set to `'on'` to have `fminunc` use a user-defined gradient of the objective function. The default `'off'` causes `fminunc` to estimate gradients using finite differences. You must provide the gradient, and set `GradObj` to `'on'`, to use the trust-region algorithm. This option is not required for the quasi-Newton algorithm.

LargeScale If you use optimoptions (recommended, see "Choose Between optimoptions and optimset" on page 2-57), use Algorithm instead of LargeScale.	Choose the algorithm. When set to the default 'on', use the 'trust-region' algorithm if possible. When set to 'off', use the 'quasi-newton' algorithm. The 'trust-region' algorithm requires you to provide the gradient (see the description of fun). Otherwise, fminunc uses the 'quasi-newton' algorithm. For more information, see "Choosing the Algorithm" on page 2-7.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default value is 100*numberOfVariables. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default value is 400. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none ([ ]). See "Output Function" on page 13-20.
PlotFcns	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none ([ ]). • @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 optimality measure. For information on writing a custom plot function, see "Plot Functions" on page 13-29.

TolFun	Termination tolerance on the function value, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on x, a positive scalar. The default value is 1e-6. See "Tolerances and Stopping Criteria" on page 2-61.
TypicalX	Typical x values. The number of elements in TypicalX is equal to the number of elements in x0, the starting point. The default value is ones(numberofvariables,1). fminunc uses TypicalX for scaling finite differences for gradient estimation.
	The trust-region algorithm uses TypicalX only for the DerivativeCheck option.

trust-region Algorithm

Hessian	If set to 'off' (default), fminunc approximates the Hessian using finite differences.
	If set to 'on', fminunc uses a user-defined Hessian for the objective function. The Hessian is either defined in the objective function (see fun), or is indirectly defined when using HessMult.
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 is 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. The matrix `W = H*Y`, although `H` is not formed explicitly. `fminunc` uses `Hinfo` to compute the preconditioner. For information on how to supply values for any additional parameters `hmfun` needs, see "Passing Extra Parameters" on page 2-50.

Note `'Hessian'` must be set to `'on'` for `fminunc` to pass `Hinfo` from `fun` to `hmfun`.

For an example, see "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75.

HessPattern	Sparsity pattern of the Hessian for finite differencing. Set HessPattern(i,j) = 1 when you can have ∂^2fun/∂x(i)∂x(j) $\neq 0$. Otherwise, set HessPattern(i,j) = O.
	Use HessPattern when it is inconvenient to compute the Hessian matrix H in fun, but you can determine (say, by inspection) when the ith component of the gradient of fun depends on x(j). fminunc can approximate H via sparse finite differences (of the gradient) if you provide the *sparsity structure* of H as the value for HessPattern. In other words, provide the locations of the nonzeros.
	When the structure is unknown, do not set HessPattern. The default behavior is as if HessPattern is a dense matrix of ones. Then fminunc computes a full finite-difference approximation in each iteration. This computation can be expensive for large problems, so it is usually better to determine the sparsity structure.
MaxPCGIter	Maximum number of preconditioned conjugate gradient (PCG) iterations, a positive scalar. The default is max(1,floor(numberOfVariables/2)). For more information, see "Trust Region Algorithm" on page 14-104.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG, a nonnegative integer. By default, fminunc uses diagonal preconditioning (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, a positive scalar. The default is O.1.

quasi-newton Algorithm

HessUpdate	Method for choosing the search direction in the Quasi-Newton algorithm. The choices are:

- `'bfgs'`, the default
- `'dfp'`
- `'steepdesc'`

  See "Hessian Update" on page 6-9 for a description of these methods.

InitialHessMatrix This option will be removed in a future release.	Initial quasi-Newton matrix. This option is available only if you set InitialHessType to `'user-supplied'`. In that case, you can set InitialHessMatrix to one of the following:

- A positive scalar — The initial matrix is the scalar times the identity.
- A vector of positive values — The initial matrix is a diagonal matrix with the entries of the vector on the diagonal. This vector should be the same size as the x0 vector, the initial point.

InitialHessType This option will be removed in a future release.	Initial quasi-Newton matrix type. The options are:

- `'identity'`
- `'scaled-identity'`, the default
- `'user-supplied'` — See InitialHessMatrix

ObjectiveLimit	A tolerance (stopping criterion) that is a scalar. If the objective function value at an iteration is less than or equal to ObjectiveLimit, the iterations halt because the problem is presumably unbounded. The default value is -1e20.

Example: `options = optimoptions('fminunc','GradObj','on')`

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
objective	Objective function

Field Name	Entry
x0	Initial point for x
solver	'fminunc'
options	Options created with optimoptions

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

Output Arguments

x — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see "When the Solver Succeeds" on page 4-22.

`fval` — Objective function value at solution
real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

`exitflag` — Reason `fminunc` stopped
integer

Reason `fminunc` stopped, returned as an integer.

1	Magnitude of gradient is smaller than the `TolFun` tolerance.
2	Change in x was smaller than the `TolX` tolerance.
3	Change in the objective function value was less than the `TolFun` tolerance.
5	Predicted decrease in the objective function was less than the `TolFun` tolerance.

0	Number of iterations exceeded `MaxIter` or number of function evaluations exceeded `MaxFunEvals`.
-1	Algorithm was terminated by the output function.
-3	Objective function at current iteration went below `ObjectiveLimit`.

output — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

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 (`'trust-region'` algorithm only)
lssteplength	Size of line search step relative to search direction (`'quasi-newton'` algorithm only)
stepsize	Final displacement in `x`
message	Exit message

grad — Gradient at the solution
real vector

Gradient at the solution, returned as a real vector. `grad` gives the gradient of `fun` at the point `x(:)`.

hessian — Approximate Hessian
real matrix

Approximate Hessian, returned as a real matrix. For the meaning of `hessian`, see "Hessian" on page 3-26.

More About

Algorithms

Trust Region Algorithm

The `trust-region` algorithm requires that you supply the gradient in fun and set `GradObj` to `'on'` using `optimoptions`. 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 "`fminunc trust-region` Algorithm" on page 6-2, "Trust-Region Methods for Nonlinear Minimization" on page 6-2 and "Preconditioned Conjugate Gradient Method" on page 6-4.

Quasi-Newton Algorithm

The `quasi-newton` algorithm 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 `Algorithm` option to `'quasi-newton'`). You can select a steepest descent method by setting `HessUpdate` to `'steepdesc'` (and `Algorithm` to `'quasi-newton'`), although this setting is usually inefficient. See "`fminunc quasi-newton` Algorithm" on page 6-5.

· "Optimization Problem Setup"
· "Unconstrained Nonlinear Optimization Algorithms" on page 6-2

References

[1] Broyden, C. G. "The Convergence of a Class of Double-Rank Minimization Algorithms." *Journal Inst. Math. Applic.*, Vol. 6, 1970, pp. 76–90.

[2] Coleman, T. F. and Y. Li. "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds." *SIAM Journal on Optimization*, Vol. 6, 1996, pp. 418–445.

[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, 1994, pp. 189–224.

[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, 1970, pp. 317–322.

[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, 1963, pp. 163–168.

[8] Goldfarb, D. "A Family of Variable Metric Updates Derived by Variational Means." *Mathematics of Computing*, Vol. 24, 1970, pp. 23–26.

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

See Also
fmincon | fminsearch | optimoptions

Introduced before R2006a

fseminf

Find minimum of semi-infinitely constrained multivariable nonlinear function

Equation

Finds the minimum of a problem specified by

$$\min_x f(x) \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub, \\ c(x) \leq 0, \\ ceq(x) = 0, \\ K_i(x, w_i) \leq 0, \ 1 \leq i \leq n. \end{cases}$$

b and beq 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) $K_i(x, w_i) \leq 0$ 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.

x, lb, and ub can be passed as vectors or matrices; see "Matrix Arguments" on page 2-30.

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 $w_i \in \Re^1$ (or $w_i \in \Re^2$). Because it is impossible to calculate all possible values of $K_i(x,w_i)$, a region must be chosen for w_i over which to calculate an appropriately sampled set of values.

Note: "Passing Extra Parameters" on page 2-50 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`.

Note: See "Iterations Can Violate Constraints" on page 2-32.

`x = fseminf(fun,x0,ntheta,seminfcon,A,b,Aeq,beq,lb,ub,options)` minimizes with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fseminf(problem)` finds the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 14-108.

Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[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 13-2 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 a file

                ```
                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 = optimoptions('fseminf','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 14-112 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 ith row of S contains the recommended sampling interval for evaluating Ki. When Ki is a vector, use only S(i,1) (the second column can be all zeros). When Ki is a matrix, S(i,2) is used for the sampling of the rows in Ki, S(i,1) is used for the sampling interval of the columns of Ki (see "Two-Dimensional Semi-Infinite Constraint" on page 6-96). 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 2-50 explains how to parameterize seminfcon, if necessary. "Example of Creating Sampling Points" on page 6-42 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 created with optimoptions

Output Arguments

"Function Arguments" on page 13-2 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.MaxFunEvals`.
	-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 constraint functions
	firstorderopt	Measure of first-order optimality
	message	Exit message

Options

Optimization options used by `fseminf`. Use `optimoptions` to set or change `options`. See "Optimization Options Reference" on page 13-7 for detailed information.

DerivativeCheck	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing

	derivatives. The choices are `'on'` or the default `'off'`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. The choices are `'on'` or the default `'off'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
Display	Level of display (see "Iterative Display" on page 3-16):

- `'off'` or `'none'` displays no output.
- `'iter'` displays output at each iteration, and gives the default exit message.
- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.
- `'notify'` displays output only if the function does not converge, and gives the default exit message.
- `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message.
- `'final'` (default) displays just the final output, and gives the default exit message.
- `'final-detailed'` displays just the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set FinDiffRelStep to a vector v, forward finite differences steps delta are delta = v.*sign ´(x).*max(abs(x),TypicalX); where sign´(x) = sign(x) except sign´(0) = 1. Central finite differences are delta = v.*max(abs(x),TypicalX); Scalar FinDiffRelStep expands to a vector. The default is sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences.
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 could 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. The default 'off' 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. Set to 'on' to have fseminf use a user-defined gradient of the objective function. The default 'off' causes fseminf to estimate gradients using finite differences.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is 100*numberOfVariables. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.

MaxIter	Maximum number of iterations allowed, a positive integer. The default is `400`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxSQPIter	Maximum number of SQP iterations allowed, a positive integer. The default is `10*max(numberOfVariables, numberOfInequalities + numberOfBounds)`.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`):

PlotFcns (continued):

- `@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 optimality measure.

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

| 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 fseminf considers too large. The default is no bounds (`[ ]`). |

`RelLineSrchBndDuration`	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is `1`)
`TolCon`	Termination tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
`TolConSQP`	Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is `1e-6`.
`TolFun`	Termination tolerance on the function value, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-61.
`TolX`	Termination tolerance on `x`, a positive scalar. The default value is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-61.
`TypicalX`	Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fseminf` uses `TypicalX` for scaling finite differences for gradient estimation.

Notes

The optimization routine `fseminf` might vary the recommended sampling interval, `S`, set in `seminfcon`, during the computation because values other than the recommended interval might be more appropriate for efficiency or robustness. Also, the finite region w_i, over which $K_i(x,w_i)$ is calculated, is allowed to vary during the optimization, provided that it does not result in significant changes in the number of local minima in $K_i(x,w_i)$.

Examples

This example minimizes the function
$(x-1)^2$,

subject to the constraints

$0 \le x \le 2$

$g(x, t) = (x - 1/2) - (t - 1/2)^2 \le 0$ for all $0 \le t \le 1$.

The unconstrained objective function is minimized at $x = 1$. However, the constraint, $g(x, t) \le 0$ 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

$\max_t g(x, t) \le 0$ 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)

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is
non-decreasing in feasible directions, to within the
default value of the function tolerance, and constraints
```

```
are satisfied to within the default value of the
constraint tolerance.

Active inequalities (to within options.TolCon = 1e-006):
   lower      upper     ineqlin    ineqnonlin
                                       1


x =
    0.5000
```

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.

More About

Algorithms

`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 `optimoptions`, see also "SQP Implementation" on page 6-29. For more details on the `fseminf` algorithm, see "fseminf Problem Formulation and Algorithm" on page 6-41.

- "Create Function Handle"

- "fseminf Problem Formulation and Algorithm" on page 6-41
- "Constrained Optimization"

See Also
`fmincon` | `optimoptions` | `optimtool`

fsolve

Solve system of nonlinear equations

Nonlinear system solver

Solves a problem specified by
$F(x) = 0$

for x, where $F(x)$ is a function that returns a vector value.

x is a vector or a matrix; see "Matrix Arguments" on page 2-30.

Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
x = fsolve(problem)
[x,fval] = fsolve( ___ )
[x,fval,exitflag,output] = fsolve( ___ )
[x,fval,exitflag,output,jacobian] = fsolve( ___ )
```

Description

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations `fun(x) = 0`, an array of zeros.

`x = fsolve(fun,x0,options)` solves the equations with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fsolve(problem)` solves `problem`, where `problem` is a structure described in "Input Arguments" on page 14-129. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[x,fval] = fsolve( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fsolve( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fsolve`, and a structure `output` with information about the optimization process.

`[x,fval,exitflag,output,jacobian] = fsolve( ___ )` returns the Jacobian of `fun` at the solution `x`.

Examples

Solution of 2-D Nonlinear System

This example shows how to solve two nonlinear equations in two variables. The equations are

$$e^{-e^{-(x_1+x_2)}} = x_2 \left(1 + x_1^2\right)$$
$$x_1 \cos\left(x_2\right) + x_2 \sin\left(x_1\right) = \tfrac{1}{2}.$$

Convert the equations to the form $F(x) = \mathbf{0}$.

$$e^{-e^{-(x_1+x_2)}} - x_2 \left(1 + x_1^2\right) = 0$$
$$x_1 \cos\left(x_2\right) + x_2 \sin\left(x_1\right) - \tfrac{1}{2} = 0.$$

Write a function that computes the left-hand side of these two equations.

```
% Copyright 2015 The MathWorks, Inc.

function F = root2d(x)

F(1) = exp(-exp(-(x(1)+x(2)))) - x(2)*(1+x(1)^2);
F(2) = x(1)*cos(x(2)) + x(2)*sin(x(1)) - 0.5;
```

Save this code as a file named `root2d.m` on your MATLAB® path.

Solve the system of equations starting at the point `[0,0]`.

```
fun = @root2d;
```

```
x0 = [0,0];
x = fsolve(fun,x0)
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

```
x =

    0.3931    0.3366
```

Solution with Nondefault Options

Examine the solution process for a nonlinear system.

Set options to have no display and a plot function that displays the first-order optimality, which should converge to 0 as the algorithm iterates.

```
options = optimoptions('fsolve','Display','none','PlotFcns',@optimplotfirstorderopt);
```

The equations in the nonlinear system are

$$e^{-e^{-(x_1+x_2)}} = x_2 \left(1 + x_1^2\right)$$
$$x_1 \cos(x_2) + x_2 \sin(x_1) = \tfrac{1}{2}.$$

Convert the equations to the form $F(x) = \mathbf{0}$.

$$e^{-e^{-(x_1+x_2)}} - x_2 \left(1 + x_1^2\right) = 0$$
$$x_1 \cos(x_2) + x_2 \sin(x_1) - \tfrac{1}{2} = 0.$$

Write a function that computes the left-hand side of these two equations.

```
% Copyright 2015 The MathWorks, Inc.

function F = root2d(x)

F(1) = exp(-exp(-x(1)+x(2))) - x(2)*(1+x(1)^2);
F(2) = x(1)*cos(x(2)) + x(2)*sin(x(1)) - 0.5;
```

Save this code as a file named root2d.m on your MATLAB® path.

Solve the nonlinear system starting from the point [0,0] and observe the solution process.

```
fun = @root2d;
x0 = [0,0];
x = fsolve(fun,x0,options)
```

```
x =

    0.3931    0.3366
```
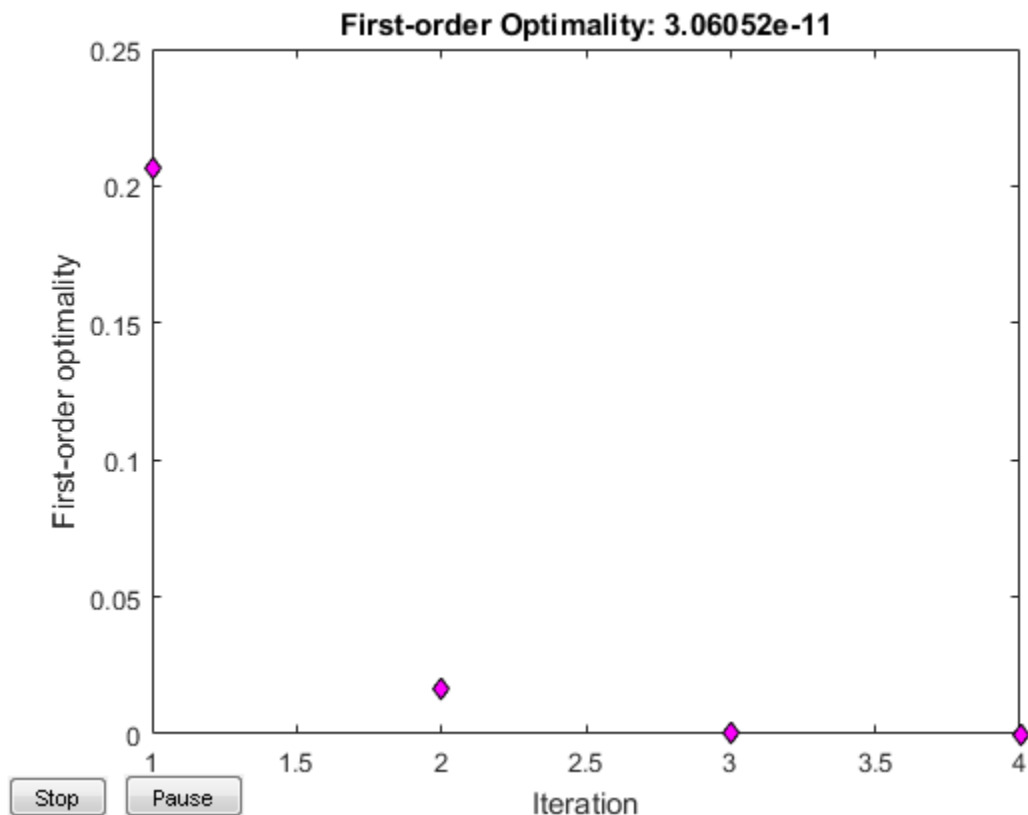
Solve a Problem Structure

Create a problem structure for `fsolve` and solve the problem.

Solve the same problem as in "Solution with Nondefault Options", but formulate the problem using a problem structure.

Set options for the problem to have no display and a plot function that displays the first-order optimality, which should converge to 0 as the algorithm iterates.

```
problem.options = optimoptions('fsolve','Display','none','PlotFcns',@optimplotfirstorde
```

The equations in the nonlinear system are

$$e^{-e^{-(x_1+x_2)}} = x_2 \left(1 + x_1^2\right)$$
$$x_1 \cos{(x_2)} + x_2 \sin{(x_1)} = \tfrac{1}{2}.$$

Convert the equations to the form $F(x) = \mathbf{0}$.

$$e^{-e^{-(x_1+x_2)}} - x_2 \left(1 + x_1^2\right) = 0$$
$$x_1 \cos{(x_2)} + x_2 \sin{(x_1)} - \tfrac{1}{2} = 0.$$

Write a function that computes the left-hand side of these two equations.

```
% Copyright 2015 The MathWorks, Inc.

function F = root2d(x)

F(1) = exp(-exp(-x(1)+x(2))) - x(2)*(1+x(1)^2);
F(2) = x(1)*cos(x(2)) + x(2)*sin(x(1)) - 0.5;
```

Save this code as a file named root2d.m on your MATLAB® path.

Create the remaining fields in the problem structure.

```
problem.objective = @root2d;
problem.x0 = [0,0];
problem.solver = 'fsolve';
```

Solve the problem.

```
x = fsolve(problem)
```

```
x =

    0.3931    0.3366
```
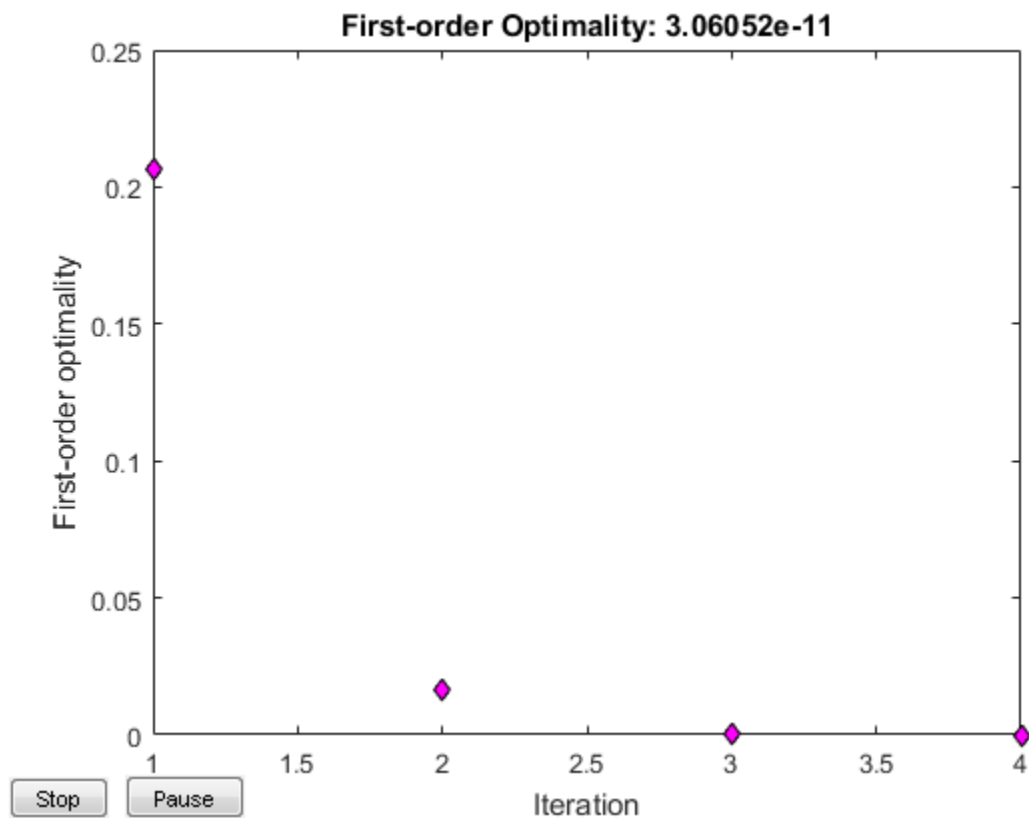
Solution Process of Nonlinear System

This example returns the iterative display showing the solution process for the system of two equations and two unknowns

$$2x_1 - x_2 = e^{-x_1}$$
$$-x_1 + 2x_2 = e^{-x_2}.$$

Rewrite the equations in the form $F(x) = 0$:

$$2x_1 - x_2 - e^{-x_1} = 0$$

$$-x_1 + 2x_2 - e^{-x_2} = 0.$$

Start your search for a solution at `x0 = [-5 -5]`.

First, write a 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))];
```

Save this function file as `myfun.m` on your MATLAB path.

Set up the initial point. Set options to return iterative display.

```
x0 = [-5;-5];
options = optimoptions('fsolve','Display','iter');
```

Call `fsolve`.

```
[x,fval] = fsolve(@myfun,x0,options)
```

```
                                 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
```

```
Equation solved.
```

```
fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

```
x =

    0.5671
```

```
      0.5671

fval =
  1.0e-006 *
      -0.4059
      -0.4059
```

Examine Matrix Equation Solution

Find a matrix X that satisfies

$$X * X * X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix},$$

starting at the point x= [1,1;1,1]. Examine the fsolve outputs to see the solution quality and process.

Create an anonymous function that calculates the matrix equation.

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

Set options to turn off the display. Set the initial point x0.

```
options = optimoptions('fsolve','Display','off');
x0 = ones(2);
```

Call fsolve and obtain information about the solution process.

```
[x,fval,exitflag,output] = fsolve(fun,x0,options)

x =

   -0.1291    0.8602
    1.2903    1.1612


fval =

    1.0e-09 *

   -0.1621    0.0780
    0.1167   -0.0465


exitflag =
```

```
        1


output =

        iterations: 6
         funcCount: 35
         algorithm: 'trust-region-dogleg'
     firstorderopt: 2.4488e-10
           message: 'Equation solved.

fsolve completed because the vector of function...'
```

The exit flag value 1 indicates that the solution is reliable. To verify this manually, calculate the residual (sum of squares of `fval`) to see how close it is to zero.

```
sum(sum(Fval.*Fval))

ans =
   4.8133e-20
```

This small residual confirms that `x` is a solution.

`fsolve` performed 35 function evaluations to find the solution, as you can see in the `output` structure.

```
output.funcCount

ans =

    35
```

- "Nonlinear Equations with Analytic Jacobian" on page 11-9
- "Nonlinear Equations with Finite-Difference Jacobian" on page 11-12
- "Nonlinear Equations with Jacobian" on page 11-14
- "Nonlinear Equations with Jacobian Sparsity Pattern" on page 11-17
- "Nonlinear Systems with Constraints" on page 11-20

Input Arguments

`fun` — Nonlinear equations to solve
function handle | function name

Nonlinear equations to solve, specified as a function handle or function name. `fun` is a function that accepts a vector `x` and returns a vector `F`, the nonlinear equations evaluated at `x`. The equations to solve are `F = 0` for all components of `F`. The function `fun` can be specified as a function handle for a file

```
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 = optimoptions('fsolve','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`.)

Example: `fun = @(x)x*x*x-[1,2;3,4]`

Data Types: `char | function_handle`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. `fsolve` uses the number of elements in and size of `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

options — Optimization options
output of `optimoptions` | structure as `optimset` returns

Optimization options, specified as the output of `optimoptions` or a structure as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose between `'trust-region-dogleg'` (default), `'trust-region-reflective'`, and `'levenberg-marquardt'`. Set the initial Levenberg-Marquardt parameter λ by setting `Algorithm` to a cell array such as `{'levenberg-marquardt',.005}`. The default $\lambda = 0.01$.
	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`. Similarly, for the trust-region-dogleg algorithm, the number of equations must be the same as the length of `x`. `fsolve` uses the Levenberg-Marquardt algorithm when the selected algorithm is unavailable. For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
DerivativeCheck	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. The choices are `'on'` or the default `'off'`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. The choices are `'on'` or the default `'off'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
Display	Level of display (see "Iterative Display" on page 3-16):
	• `'off'` or `'none'` displays no output.

- `'iter'` displays output at each iteration, and gives the default exit message.

- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.

- `'final'` (default) displays just the final output, and gives the default exit message.

- `'final-detailed'` displays just the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are
	`delta = v.*sign′(x).*max(abs(x),TypicalX);`
	where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are
	`delta = v.*max(abs(x),TypicalX);`
	Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.

FinDiffType	Finite differences, used to estimate gradients, are either `'forward'` (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 could take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
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`. The default, `'off'`, 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'` (default), `fsolve` approximates the Jacobian using finite differences.

MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default is `400`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`):

- `@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 optimality measure.

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on `x`, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.

TypicalX	Typical x values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fsolve` uses `TypicalX` for scaling finite differences for gradient estimation.
	The `trust-region-dogleg` algorithm uses `TypicalX` as the diagonal terms of a scaling matrix.

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 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, in

```
[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 2-50 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'Jacobian'` must be set to `'on'` for `fsolve` to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75 for a similar example.

JacobPattern	Sparsity pattern of the Jacobian for finite differencing. Set `JacobPattern(i,j) = 1` when `fun(i)` depends on `x(j)`. Otherwise, set `JacobPattern(i,j) = 0`. In other words, `JacobPattern(i,j) = 1` when you can have ∂`fun(i)`/∂`x(j)` $\neq 0$.
	Use `JacobPattern` when it is inconvenient to compute the Jacobian matrix `J` in `fun`, though you can determine (say, by inspection) when `fun(i)` depends on `x(j)`. `fsolve` can approximate `J` via sparse finite differences when you give `JacobPattern`.
	In the worst case, if the structure is unknown, do not set `JacobPattern`. The default behavior is as if `JacobPattern` is a dense matrix of ones. Then `fsolve` computes a full finite-difference approximation in each iteration. This can be very expensive for large problems, so it is usually better to determine the sparsity structure.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,floor(numberOfVariables/2))`. For more information, see "Equation Solving Algorithms" on page 11-2.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG, a nonnegative integer. 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, a positive scalar. The default is `0.1`.

Levenberg-Marquardt Algorithm

InitDamping	Initial value of the Levenberg-Marquardt parameter, a positive scalar. Default is `1e-2`. For details, see "Levenberg-Marquardt Method" on page 10-7.

| ScaleProblem | 'Jacobian' can sometimes improve the convergence of a poorly scaled problem. The default is 'none'. |

Example: options = optimoptions('fsolve','Jacobian','on')

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
objective	Objective function
x0	Initial point for x
solver	'fsolve'
options	Options created with optimoptions

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: struct

Output Arguments

x — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when exitflag is positive. For information on the quality of the solution, see "When the Solver Succeeds" on page 4-22.

fval — Objective function value at the solution
real vector

Objective function value at the solution, returned as a real vector. Generally, fval = fun(x).

exitflag — Reason fsolve stopped
integer

Reason `fsolve` stopped, returned as an integer.

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.MaxFunEvals`.
-1	Output function terminated the algorithm.
-2	Algorithm appears to be converging to a point that is not a root.
-3	Trust region radius became too small (`trust-region-dogleg` algorithm).
-4	Line search cannot sufficiently decrease the residual along the current search direction.

output — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations
`algorithm`	Optimization algorithm used
`cgiterations`	Total number of PCG iterations (`'trust-region-reflective'` algorithm only)
`stepsize`	Final displacement in `x` (not in `'trust-region-dogleg'`)
`firstorderopt`	Measure of first-order optimality
`message`	Exit message

jacobian — Jacobian at the solution
real matrix

Jacobian at the solution, returned as a real matrix. `jacobian(i,j)` is the partial derivative of `fun(i)` with respect to `x(j)` at the solution `x`.

Limitations

- The function to be solved must be continuous.
- When successful, `fsolve` only gives one root.
- 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 method, the system of equations need not be square.
- The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective algorithm 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 $J^T J$, might lead to a costly solution process for large problems.

More About

Algorithms

The 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 14-139).

- 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]. See "Trust-Region Dogleg Method" on page 11-5.
- 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 11-2.
- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 11-7.

- "Optimization Problem Setup"
- "Equation Solving Algorithms" on page 11-2

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

`fzero` | `lsqcurvefit` | `lsqnonlin` | `optimoptions`

Introduced before R2006a

fzero

Root of nonlinear function

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)

x = fzero(problem)

[x,fval,exitflag,output] = fzero( ___ )
```

Description

`x = fzero(fun,x0)` tries to find a point x where `fun(x) = 0`. This solution is where `fun(x)` changes sign—`fzero` cannot find a root of a function such as `x^2`.

`x = fzero(fun,x0,options)` uses `options` to modify the solution process.

`x = fzero(problem)` solves a root-finding problem specified by `problem`.

`[x,fval,exitflag,output] = fzero( ___ )` returns `fun(x)` in the `fval` output, `exitflag` encoding the reason `fzero` stopped, and an output structure containing information on the solution process.

Examples

Root Starting From One Point

Calculate π by finding the zero of the sine function near 3.

```
fun = @sin; % function
x0 = 3; % initial point
x = fzero(fun,x0)
```

```
x =

    3.1416
```

Root Starting From an Interval

Find the zero of cosine between 1 and 2.

```
fun = @cos; % function
x0 = [1 2]; % initial interval
x = fzero(fun,x0)
```

```
x =

    1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

Root of a Function Defined by a File

Find a zero of the function $f(x) = x^3 - 2x - 5$.

First, write a file called `f.m`.

```
function y = f(x)
y = x.^3-2*x-5;
```

Save `f.m` on your MATLAB path.

Find the zero of $f(x)$ near 2.

```
fun = @f; % function
x0 = 2; % initial point
z = fzero(fun,x0)
```

```
z =
    2.0946
```

Since `f(x)` is a polynomial, you can find the same real zero, and a complex conjugate pair of zeros, using the `roots` command.

```
roots([1 0 -2 -5])

  ans =
  2.0946
 -1.0473 + 1.1359i
 -1.0473 - 1.1359i
```

Root of Function with Extra Parameter

Find the root of a function that has an extra parameter.

```
myfun = @(x,c) cos(c*x);   % parameterized function
c = 2;                     % parameter
fun = @(x) myfun(x,c);     % function of x alone
x = fzero(fun,0.1)


x =

    0.7854
```

Nondefault Options

Plot the solution process by setting some plot functions.

Define the function and initial point.

```
fun = @(x)sin(cosh(x));
x0 = 1;
```

Examine the solution process by setting options that include plot functions.

```
options = optimset('PlotFcns',{@optimplotx,@optimplotfval});
```

Run fzero including options.

```
x = fzero(fun,x0,options)


x =

    1.8115
```
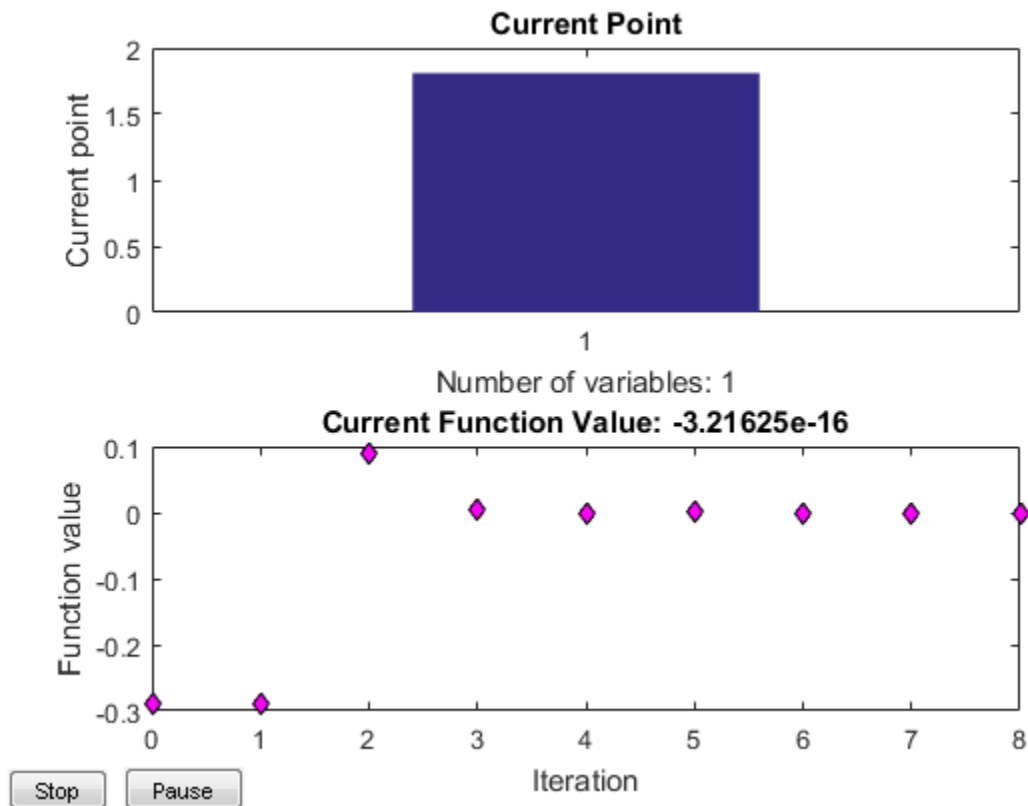
Solve Exported Problem

Solve a problem that is defined by an export from Optimization app.

Define a problem in Optimization app. Enter `optimtool('fzero')`, and fill in the problem as pictured.

Note: The Optimization app warns that it will be removed in a future release.

Select File > Export to Workspace, and export the problem as pictured to a variable named problem.

Enter the following at the command line.

```
x = fzero(problem)

x =

   1.8115
```

More Information from Solution

Find the point where exp(-exp(-x)) = x, and display information about the solution process.

```
fun = @(x) exp(-exp(-x)) - x; % function
x0 = [0,1]; % initial interval
options = optimset('Display','iter'); % show iterations
[x,fval,exitflag,output] = fzero(fun,x0,options)


 Func-count     x          f(x)             Procedure
    2            1        -0.307799         initial
    3        0.544459      0.0153522        interpolation
    4        0.566101      0.00070708       interpolation
    5        0.567143     -1.40255e-08      interpolation
    6        0.567143      1.50013e-12      interpolation
    7        0.567143             0         interpolation

Zero found in the interval [0, 1]

x =

    0.5671


fval =

     0


exitflag =

     1


output =

    intervaliterations: 0
            iterations: 5
             funcCount: 7
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [0, 1]'
```

`fval = 0` means `fun(x) = 0`, as desired.

· "Roots of Scalar Functions"

Input Arguments

fun — Function to solve
function handle

Function to solve, specified as a handle to a scalar-valued function. `fun` accepts a scalar `x` and returns a scalar `fun(x)`.

`fzero` solves `fun(x) = 0`. To solve an equation `fun(x) = c(x)`, instead solve `fun2(x) = fun(x) - c(x) = 0`.

To include extra parameters in your function, see the example "Root of Function with Extra Parameter" on page 14-143 and the section "Passing Extra Parameters" on page 2-50.

Example: `@sin`

Example: `@myFunction`

Example: `@(x)(x-a)^5 - 3*x + a - 1`

Data Types: `function_handle`

x0 — Initial value
scalar | 2-element vector

Initial value, specified as a real scalar or a 2-element real vector.

- Scalar — `fzero` begins at `x0` and tries to locate a point `x1` where `fun(x1)` has the opposite sign of `fun(x0)`. Then `fzero` iteratively shrinks the interval where `fun` changes sign to reach a solution.
- 2-element vector — `fzero` checks that `fun(x0(1))` and `fun(x0(2))` have opposite signs, and errors if they do not. It then iteratively shrinks the interval where `fun` changes sign to reach a solution. An interval `x0` must be finite; it cannot contain ±`Inf`.

Tip Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

Example: 3

Example: [2,17]

Data Types: `double`

`options` — Options for solution process
structure, typically created using `optimset`

Options for solution process, specified as a structure. Create or modify the `options` structure using `optimset`. `fzero` uses these `options` structure fields.

Display	Level of display (see "Iterative Display" on page 3-16):

     • `'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`, `Inf`, or `NaN`.

     • The default, `'off'`, displays no error.

OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See "Output Function" on page 13-20.
PlotFcns	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`).

     • `@optimplotx` plots the current point.

     • `@optimplotfval` plots the function value.

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

TolX	Termination tolerance on x, a positive scalar. The default is `eps`, 2.2204e–16. See "Tolerances and Stopping Criteria" on page 2-61.

Example: `options = optimset('FunValCheck','on')`

Data Types: `struct`

`problem` — Root-finding problem

structure

Root-finding problem, specified as a structure with all of the following fields.

objective	Objective function
x0	Initial point for x, scalar or 2-D vector
solver	'fzero'
options	Options structure, typically created using optimset

You can generate `problem` by exporting from Optimization app. See "Importing and Exporting Your Work" on page 5-11 or "Solve Exported Problem" on page 14-144.

Data Types: struct

Output Arguments

`x` — Location of root or sign change

real scalar

Location of root or sign change, returned as a scalar.

`fval` — Function value at `x`

real scalar

Function value at x, returned as a scalar.

`exitflag` — Integer encoding the exit condition

integer

Integer encoding the exit condition, meaning the reason fzero stopped its iterations.

1	Function converged to a solution x.
-1	Algorithm was terminated by the output function or plot function.
-3	NaN or Inf function value was encountered while searching for an interval containing a sign change.
-4	Complex function value was encountered while searching for an interval containing a sign change.

`-5`	Algorithm might have converged to a singular point.
`-6`	`fzero` did not detect a sign change.

`output` — Information about root-finding process
structure

Information about root-finding process, returned as a structure. 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`	`'bisection, interpolation'`
`message`	Exit message

More About

Algorithms

The `fzero` command is a function file. The algorithm, created 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 `fzero` is based, is in [2].

- "Passing Extra Parameters" on page 2-50

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
fminbnd | fsolve | optimset | optimtool | roots

Introduced before R2006a

fzmult

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

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`

intlinprog

Mixed-integer linear programming (MILP)

Mixed-integer linear programming solver.

Finds the minimum of a problem specified by

$$\min_{x} f^T x \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub. \end{cases}$$

f, *x*, intcon, *b*, *beq*, *lb*, and *ub* are vectors, and *A* and *Aeq* are matrices.

You can specify *f*, intcon, *lb*, and *ub* as vectors or arrays. See "Matrix Arguments" on page 2-30.

Syntax

```
x = intlinprog(f,intcon,A,b)
x = intlinprog(f,intcon,A,b,Aeq,beq)
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,options)
x = intlinprog(problem)
[x,fval,exitflag,output] = intlinprog( ___ )
```

Description

`x = intlinprog(f,intcon,A,b)` solves min `f'*x` such that the components of `x` in `intcon` are integers, and `A*x ≤ b`.

`x = intlinprog(f,intcon,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 = intlinprog(f,intcon,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 = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,options)` minimizes using the optimization options specified in `options`. Use `optimoptions` to set these options. Set `lb = []` and `ub = []` if no bounds exist.

`x = intlinprog(problem)` uses a `problem` structure to encapsulate all solver inputs. You can import a `problem` structure from an MPS file using `mpsread`.

`[x,fval,exitflag,output] = intlinprog( ___ )`, for any input arguments described above, returns `fval = f'*x`, a value `exitflag` describing the exit condition, and a structure `output` containing information about the optimization process.

Examples

Solve an MILP with Linear Inequalities

Solve the problem

$$\min_x 8x_1 + x_2 \text{ subject to } \begin{cases} x_2 \text{ is an integer} \\ x_1 + 2x_2 \geq -14 \\ -4x_1 - x_2 \leq -33 \\ 2x_1 + x_2 \leq 20. \end{cases}$$

Write the objective function vector and vector of integer variables.

```
f = [8;1];
intcon = 2;
```

Convert all inequalities into the form `A*x <= b` by multiplying "greater than" inequalities by `-1`.

```
A = [-1,-2;
     -4,-1;
      2,1];
b = [14;-33;20];
```

Call `intlinprog`.

```
x = intlinprog(f,intcon,A,b)
```

```
LP:                 Optimal objective value is 59.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerar
options.TolGapAbs = 0 (the default value). The intcon variables are integer within tole
options.TolInteger = 1e-05 (the default value).


x =

    6.5000
    7.0000
```

Solve an MILP with All Types of Constraints

Solve the problem

$$\min_x\left(-3x_1 - 2x_2 - x_3\right) \text{ subject to } \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

Write the objective function vector and vector of integer variables.

```
f = [-3;-2;-1];
intcon = 3;
```

Write the linear inequality constraints.

```
A = [1,1,1];
b = 7;
```

Write the linear equality constraints.

```
Aeq = [4,2,1];
beq = 12;
```

Write the bound constraints.

```
lb = zeros(3,1);
ub = [Inf;Inf;1]; % Enforces x(3) is binary
```

Call intlinprog.

```
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)
```

```
LP:                Optimal objective value is -12.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap toleran
options.TolGapAbs = 0 (the default value). The intcon variables are integer within tole
options.TolInteger = 1e-05 (the default value).


x =

         0
    5.5000
    1.0000
```

Solve an MILP with Nondefault Options

Solve the problem

$$\min_{x}\left(-3x_1 - 2x_2 - x_3\right) \text{ subject to } \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

without showing iterative display.

Specify the solver inputs.

```
f = [-3;-2;-1];
intcon = 3;
A = [1,1,1];
b = 7;
Aeq = [4,2,1];
beq = 12;
lb = zeros(3,1);
ub = [Inf;Inf;1]; % enforces x(3) is binary
```

Specify no display.

```
options = optimoptions('intlinprog','Display','off');
```

Run the solver.

```
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,options)

x =

         0
    5.5000
    1.0000
```

Use a Problem Structure

Solve the problem

$$\min_{x}\left(-3x_1 - 2x_2 - x_3\right) \text{ subject to } \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

using iterative display. Use a `problem` structure as the `intlinprog` input.

Specify the solver inputs.

```
f = [-3;-2;-1];
intcon = 3;
A = [1,1,1];
b = 7;
Aeq = [4,2,1];
beq = 12;
lb = zeros(3,1);
ub = [Inf;Inf;1]; % enforces x(3) is binary
options = optimoptions('intlinprog','Display','off');
```

Insert the inputs into a problem structure. Include the solver name.

```
problem = struct('f',f,'intcon',intcon,...
    'Aineq',A,'bineq',b,'Aeq',Aeq,'beq',beq,...
    'lb',lb,'ub',ub,'options',options,...
    'solver','intlinprog');
```

Run the solver.

```
x = intlinprog(problem)

x =
```

```
        0
   5.5000
   1.0000
```

Examine the MILP Solution and Process

Call `intlinprog` with more outputs to see solution details and process.

The goal is to solve the problem

$$\min_{x}\left(-3x_1 - 2x_2 - x_3\right) \text{ subject to } \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

Specify the solver inputs.

```
f = [-3;-2;-1];
intcon = 3;
A = [1,1,1];
b = 7;
Aeq = [4,2,1];
beq = 12;
lb = zeros(3,1);
ub = [Inf;Inf;1]; % enforces x(3) is binary
```

Call `intlinprog` with all outputs.

```
[x,fval,exitflag,output] = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)

LP:                Optimal objective value is -12.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerar
options.TolGapAbs = 0 (the default value). The intcon variables are integer within tole
options.TolInteger = 1e-05 (the default value).


x =

        0
```

```
      5.5000
      1.0000


fval =

   -12.0000


exitflag =

       1


output =

          relativegap: 0
          absolutegap: 0
       numfeaspoints: 1
            numnodes: 0
     constrviolation: 1.7764e-15
              message: 'Optimal solution found.
```

Intlinprog stopped at the root node because the objective v...'

The output structure shows numnodes is 0. This means intlinprog solved the problem before branching. This is one indication that the result is reliable. Also, the absolutegap and relativegap fields are 0. This is another indication that the result is reliable.

- "Mixed-Integer Linear Programming Basics" on page 8-49
- "Factory, Warehouse, Sales Allocation Model" on page 8-52
- "Travelling Salesman Problem" on page 8-64
- "Solve Sudoku Puzzles Via Integer Programming" on page 8-94
- "Mixed-Integer Quadratic Programming Portfolio Optimization" on page 8-85
- "Optimal Dispatch of Power Generators" on page 8-72

Input Arguments

f — Coefficient vector
real vector

Coefficient vector, specified as a vector of doubles representing the objective function, `f'*x`. The notation assumes that `f` is a column vector, but you are free to use a row vector.

`f` can also be an array. Internally, `intlinprog` converts an array `f` to the vector `f(:)`.

If you specify `f = []`, `intlinprog` tries to find a feasible point without trying to minimize an objective function.

Example: `f = [4;2;-1.7];`

Data Types: `double`

`intcon` — Vector of integer constraints
vector of integers

Vector of integer constraints, specified as a vector of positive integers. The values in `intcon` indicate the components of the decision variable `x` that are integer-valued. `intcon` has values from `1` through `numel(f)`.

`intcon` can also be an array. Internally, `intlinprog` converts an array `intcon` to the vector `intcon(:)`.

Example: `intcon = [1,2,7]` means `x(1)`, `x(2)`, and `x(7)` take only integer values.

Data Types: `double`

A — Linear inequality constraint matrix
real matrix

Linear inequality constraint matrix, specified as a matrix of doubles. A represents the linear coefficients in the constraints `A*x` $\leq$ `b`. A has size M-by-N, where M is the number of constraints and N = `numel(f)`. To save memory, A can be sparse.

Example: `A = [4,3;2,0;4,-1];` means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: `double`

b — Linear inequality constraint vector
real vector

Linear inequality constraint vector, specified as a vector of doubles. b represents the constant vector in the constraints `A*x` $\leq$ `b`. b has length M, where A is M-by-N.

Example: `[4,0]`

Data Types: `double`

Aeq — Linear equality constraint matrix
`[]` (default) | real matrix

Linear equality constraint matrix, specified as a matrix of doubles. `Aeq` represents the linear coefficients in the constraints `Aeq*x = beq`. `Aeq` has size `Meq`-by-N, where `Meq` is the number of constraints and `N = numel(f)`. To save memory, `Aeq` can be sparse.

Example: `A = [4,3;2,0;4,-1];` means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: `double`

beq — Linear equality constraint vector
`[]` (default) | real vector

Linear equality constraint vector, specified as a vector of doubles. `beq` represents the constant vector in the constraints `Aeq*x = beq`. `beq` has length `Meq`, where `Aeq` is `Meq`-by-N.

Example: `[4,0]`

Data Types: `double`

lb — Lower bounds
`[]` (default) | real vector or array

Lower bounds, specified as a vector or array of doubles. `lb` represents the lower bounds elementwise in `lb ≤ x ≤ ub`.

Internally, `intlinprog` converts an array `lb` to the vector `lb(:)`.

Example: `lb = [0;-Inf;4]` means `x(1) ≥ 0, x(3) ≥ 4`.

Data Types: `double`

ub — Upper bounds
`[]` (default) | real vector or array

Upper bounds, specified as a vector or array of doubles. `ub` represents the upper bounds elementwise in `lb ≤ x ≤ ub`.

Internally, `intlinprog` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means `x(2) ≤ 4, x(3) ≤ 10`.

Data Types: `double`

`options` — Options for `intlinprog`
options created using `optimoptions`

Options for `intlinprog`, specified as the output of `optimoptions`.

Option	Description	Default
BranchingRu	Rule for choosing the component for branching: • `'maxpscost'` — The fractional component with maximum pseudocost. See "Branch and Bound" on page 8-42. • `'mostfractional'` — The component whose fractional part is closest to $1/2$. • `'maxfun'` — The fractional component with maximal corresponding component in the absolute value of objective vector `f`.	`'maxpscost'`
CutGenerati	Level of cut generation (see "Cut Generation" on page 8-40): • `'none'` — No cuts. Makes `CutGenMaxIter` irrelevant. • `'basic'` — Normal cut generation. • `'intermediate'` — Use more cut types. • `'advanced'` — Use most cut types.	`'basic'`
CutGenMaxIt	Number of passes through all cut generation methods before entering the branch-and-bound phase, an integer from `1` through `50`. Disable cut generation by setting the `CutGeneration` option to `'none'`.	10
Display	Level of display (see "Iterative Display" on page 3-16): • `'off'` or `'none'` — No iterative display • `'final'` — Show final values only • `'iter'` — Show iterative display	`'iter'`

Option	Description	Default
Heuristics	Algorithm for searching for feasible points (see "Heuristics for Finding Feasible Solutions" on page 8-41): • `'none'` • `'rss'` • `'round'` • `'rins'`	`'rss'`
HeuristicsM	Strictly positive integer that bounds the number of nodes `intlinprog` can explore in its branch-and-bound search for feasible points. See "Heuristics for Finding Feasible Solutions" on page 8-41.	50
IPPreproces	Types of integer preprocessing (see "Mixed-Integer Program Preprocessing" on page 8-40): • `'none'` — Use very few integer preprocessing steps. • `'basic'` — Use a moderate number of integer preprocessing steps. • `'advanced'` — Use all available integer preprocessing steps.	`'basic'`
LPMaxIter	Strictly positive integer, the maximum number of simplex algorithm iterations per node during the branch-and-bound process.	3e4
LPPreproces	Type of preprocessing for the solution to the relaxed linear program (see "Linear Program Preprocessing" on page 8-39): • `'none'` — No preprocessing. • `'basic'` — Use preprocessing.	`'basic'`
MaxNodes	Strictly positive integer that is the maximum number of nodes `intlinprog` explores in its branch-and-bound process.	1e7

Option	Description	Default
MaxNumFeasP	Strictly positive integer. `intlinprog` stops if it finds `MaxNumFeasPoints` integer feasible points.	`Inf`
MaxTime	Positive real that is the maximum time in seconds that `intlinprog` runs.	`7200`
NodeSelecti	Choose the node to explore next. • `'simplebestproj'` — Best projection. See "Branch and Bound" on page 8-42. • `'minobj'` — Explore the node with the minimum objective function. • `'mininfeas'` — Explore the node with the minimal sum of integer infeasibilities. See "Branch and Bound" on page 8-42.	`'simplebestproj'`
ObjectiveCu	Real greater than `-Inf`. During the branch-and-bound calculation, `intlinprog` discards any node where the linear programming solution has an objective value exceeding `ObjectiveCutOff`.	`Inf`
OutputFcn	Specify one or more functions that an optimization function calls at events, either as a function handle or as a cell array of function handles. • `@savemilpsolutions` collects the integer-feasible points in the `xIntSol` matrix in your workspace, where each column is one integer feasible point. For information on writing a custom output function, see "`intlinprog` Output Functions and Plot Functions" on page 13-30.	`[]`

Option	Description	Default
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. • @optimplotmilp plots the internally-calculated upper and lower bounds on the objective value of the solution. For information on writing a custom plot function, see "intlinprog Output Functions and Plot Functions" on page 13-30.	[]
RelObjThresl	Nonnegative real. intlinprog changes the current feasible solution only when it locates another with an objective function value that is at least RelObjThreshold lower: (fold – fnew)/(1 + fold) > RelObjThreshold.	1e-4
RootLPAlgor	Algorithm for solving linear programs: • 'dual-simplex' — Dual simplex algorithm • 'primal-simplex' — Primal simplex algorithm	'dual-simplex'
RootLPMaxIt	Nonnegative integer that is the maximum number of simplex algorithm iterations to solve the initial linear programming problem.	3e4
TolCon	Real from 1e-9 through 1e-3 that is the maximum discrepancy that linear constraints can have and still be considered satisfied. TolCon is not a stopping criterion.	1e-4
TolFunLP	Nonnegative real where reduced costs must exceed TolFunLP for a variable to be taken into the basis.	1e-7

Option	Description	Default
TolGapAbs	Nonnegative real. intlinprog stops if the difference between the internally calculated upper (U) and lower (L) bounds on the objective function is less than or equal to TolGapAbs: U − L <= TolGapAbs.	0
TolGapRel	Real from 0 through 1. intlinprog stops if the relative difference between the internally calculated upper (U) and lower (L) bounds on the objective function is less than or equal to TolGapRel: (U − L) / (abs(U) + 1) <= TolGapRel.	1e-4
TolInteger	Real from 1e-6 through 1e-3, where the maximum deviation from integer that a component of the solution x can have and still be considered an integer. TolInteger is not a stopping criterion.	1e-5

Example: options = optimoptions('intlinprog','MaxTime',120)

problem — Structure encapsulating inputs and options
structure

Structure encapsulating the inputs and options, specified with the following fields.

f	Vector representing objective f'*x (required)
intcon	Vector indicating variables that take integer values (required)
Aineq	Matrix in linear inequality constraints Aineq*x ≤ bineq
bineq	Vector in linear inequality constraints Aineq*x ≤ bineq
Aeq	Matrix in linear equality constraints Aeq*x = beq
beq	Vector in linear equality constraints Aeq*x = beq
lb	Vector of lower bounds
ub	Vector of upper bounds
solver	'intlinprog' (required)

options   Options created using optimoptions (required)

You must specify at least these fields in the problem structure. Other fields are optional:

- f
- intcon
- solver
- options

Example: problem.f = [1,2,3];
problem.intcon = [2,3];
problem.options = optimoptions('intlinprog');
problem.Aineq = [-3,-2,-1];
problem.bineq = -20;
problem.lb = [-6.1,-1.2,7.3];
problem.solver = 'intlinprog';

Data Types: struct

Output Arguments

x — Solution
real vector

Solution, returned as a vector that minimizes f'*x subject to all bounds, integer constraints, and linear constraints.

When a problem is infeasible or unbounded, x is [].

fval — Objective value
real scalar

Objective value, returned as the scalar value f'*x at the solution x.

When a problem is infeasible or unbounded, fval is [].

exitflag — Algorithm stopping condition
integer

Algorithm stopping condition, returned as an integer identifying the reason the algorithm stopped. The following lists the values of `exitflag` and the corresponding reasons `intlinprog` stopped.

2	`intlinprog` stopped prematurely. Integer feasible point found.
1	`intlinprog` converged to the solution `x`.
0	`intlinprog` stopped prematurely. No integer feasible point found.
-2	No feasible point found.
-3	Root LP problem is unbounded.

The exit message can give more detailed information on the reason `intlinprog` stopped, such as exceeding a tolerance.

output — Solution process summary
structure

Solution process summary, returned as a structure containing information about the optimization process.

`relativegap`	Relative difference between upper (U) and lower (L) bounds of the objective function that `intlinprog` calculates in its branch-and-bound algorithm.
	`relativegap = (U - L) / (abs(U) + 1)`
	If `intcon = []`, `relativegap = []`.
`absolutegap`	Difference between upper and lower bounds of the objective function that `intlinprog` calculates in its branch-and-bound algorithm.
	If `intcon = []`, `absolutegap = []`.
`numfeaspoints`	Number of integer feasible points found.
	If `intcon = []`, `numfeaspoints = []`. Also, if the initial relaxed problem is infeasible, `numfeaspoints = []`.

numnodes	Number of nodes in branch-and-bound algorithm. If the solution was found during preprocessing or during the initial cuts, numnodes = 0.
	If intcon = [], numnodes = [].
constrviolation	Constraint violation that is positive for violated constraints.
	constrviolation = max([0; norm(Aeq*x-beq, inf); (lb-x); (x-ub); (Ai*x-bi)])
message	Exit message.

Limitations

- Often, some supposedly integer-valued components of the solution x(intCon) are not precisely integers. intlinprog deems as integers all solution values within the TolInteger tolerance of an integer.

  To round all supposed integers to be exactly integers, use the round function.

  ```
  x(intcon) = round(x(intcon));
  ```

  Caution Rounding solutions can cause the solution to become infeasible. Check feasibility after rounding:

  ```
  max(A*x - b) % See if entries are not too positive, so have small infeasibility
  max(abs(Aeq*x - beq)) % See if entries are near enough to zero
  max(x - ub) % Positive entries are violated bounds
  max(lb - x) % Positive entries are violated bounds
  ```

- intlinprog does not enforce that solution components be integer-valued when their absolute values exceed 2.1e9. When your solution has such components, intlinprog warns you. If you receive this warning, check the solution to see whether supposedly integer-valued components of the solution are close to integers.

- intlinprog does not allow components of the problem, such as coefficients in f, A, or ub, to exceed 1e25 in absolute value. If you try to run intlinprog with such a problem, intlinprog issues an error.

- Currently, you cannot run intlinprog in the "Optimization App" on page 5-2.

More About

MILP

Mixed-integer linear programming definition.

MILP means find the minimum of a problem specified by

$$\min_{x} f^{T}x \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub. \end{cases}$$

f, *x*, intcon, *b*, *beq*, *lb*, and *ub* are vectors, and *A* and *Aeq* are matrices.

You can specify *f*, intcon, *lb*, and *ub* as vectors or arrays. See "Matrix Arguments" on page 2-30.

Tips

- To specify binary variables, set the variables to be integers in `intcon`, and give them lower bounds of `0` and upper bounds of `1`.

- Save memory by specifying sparse linear constraint matrices `A` and `Aeq`. However, you cannot use sparse matrices for `b` and `beq`.

- To provide logical indices for integer components, meaning a binary vector with `1` indicating an integer, convert to `intcon` form using `find`. For example,

  ```
  logicalindices = [1,0,0,1,1,0,0];
  intcon = find(logicalindices)

  intcon =

       1     4     5
  ```

- `intlinprog` replaces `bintprog`. To update old `bintprog` code to use `intlinprog`, make the following changes:

  - Set `intcon` to `1:numVars`, where `numVars` is the number of variables in your problem.

  - Set `lb` to `zeros(numVars,1)`.

- Set ub to ones(numVars,1).

- Update any relevant options. Use optimoptions to create options for intlinprog.

- Change your call to bintprog as follows:

```
[x,fval,exitflag,output] = bintprog(f,A,b,Aeq,Beq,x0,options)
% Change your call to:
[x,fval,exitflag,output] = intlinprog(f,intcon,A,b,Aeq,Beq,lb,ub,options)
```

- "Mixed-Integer Linear Programming Algorithms" on page 8-38
- "Tuning Integer Linear Programming" on page 8-46
- "Optimization Problem Setup"

See Also
linprog | mpsread | optimoptions

Introduced in R2014a

linprog

Solve linear programming problems

Linear programming solver

Finds the minimum of a problem specified by

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

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,output] = linprog( ___ )
[x,fval,exitflag,output,lambda] = linprog( ___ )
```

Description

`x = linprog(f,A,b)` solves min `f'*x` such that `A*x` ≤ `b`.

`x = linprog(f,A,b,Aeq,beq)` includes 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.

Note: If the specified input bounds for a problem are inconsistent, the output x is x0 and the output fval is [].

---

x = linprog(f,A,b,Aeq,beq,lb,ub,x0) sets the starting point to x0.

Note linprog uses x0 for only the 'active-set' algorithm. For all other algorithms, linprog ignores x0.

---

x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options) minimizes with the optimization options specified by options. Use optimoptions to set these options.

x = linprog(problem) finds the minimum for problem, where problem is a structure described in "Input Arguments" on page 14-184.

Create the problem structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11. You can import a problem structure from an MPS file using mpsread.

[x,fval] = linprog(___), for any input arguments, returns the value of the objective function fun at the solution x: fval = f'*x.

[x,fval,exitflag,output] = linprog(___) additionally returns a value exitflag that describes the exit condition, and a structure output that contains information about the optimization process.

[x,fval,exitflag,output,lambda] = linprog(___) additionally returns a structure lambda whose fields contain the Lagrange multipliers at the solution x.

Examples

Linear Program, Linear Inequality Constraints

Solve a simple linear program defined by linear inequalities.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
A = [1  1
     1  1/4
     1  -1
     -1/4 -1
     -1  -1
     -1  1];
```

```
b = [2 1 2 1 -1 2];
```

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Solve the linear program.

```
x = linprog(f,A,b)
```

```
Optimization terminated.
```

```
x =

    0.6667
    1.3333
```

Linear Program with Linear Inequalities and Equalities

Solve a simple linear program defined by linear inequalities and linear equalities.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
A = [1 1
     1 1/4
     1 -1
     -1/4 -1
     -1 -1
     -1 1];

b = [2 1 2 1 -1 2];
```

Use the linear equality constraint $x(1) + x(2)/4 = 1/2$.

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

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Solve the linear program.

```
x = linprog(f,A,b,Aeq,beq)
```

```
Optimization terminated.

x =

    0.0000
    2.0000
```

Linear Program with All Constraint Types

Solve a simple linear program with linear inequalities, linear equalities, and bounds.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \le 2$$

$$x(1) + x(2)/4 \le 1$$

$$x(1) - x(2) \le 2$$

$$-x(1)/4 - x(2) \le 1$$

$$-x(1) - x(2) \le -1$$

$$-x(1) + x(2) \le 2.$$

```
A = [1 1
     1 1/4
     1 -1
     -1/4 -1
     -1 -1
     -1 1];
```

```
b = [2 1 2 1 -1 2];
```

Use the linear equality constraint $x(1) + x(2)/4 = 1/2$.

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

Set these bounds:

$$-1 \le x(1) \le 1.5$$

$$-0.5 \le x(2) \le 1.25.$$

```
lb = [-1,-0.5];
ub = [1.5,1.25];
```

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Solve the linear program.

```
x = linprog(f,A,b,Aeq,beq,lb,ub)
```

```
Optimization terminated.
```

```
x =
```

```
    0.1875
    1.2500
```

Linear Program Using the Dual-Simplex Algorithm

Solve a linear program using the `'dual-simplex'` algorithm.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \le 2$$

$$x(1) + x(2)/4 \le 1$$

$$x(1) - x(2) \le 2$$

$$-x(1)/4 - x(2) \le 1$$

$$-x(1) - x(2) \le -1$$

$$-x(1) + x(2) \le 2.$$

```
A = [1  1
     1  1/4
     1  -1
     -1/4 -1
     -1  -1
     -1  1];
```

```
b = [2 1 2 1 -1 2];
```

Use the linear equality constraint $x(1) + x(2)/4 = 1/2$.

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

Set these bounds:

$$-1 \le x(1) \le 1.5$$

$$-0.5 \le x(2) \le 1.25.$$

```
lb = [-1,-0.5];
ub = [1.5,1.25];
```

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Set options to use the `'dual-simplex'` algorithm.

```
options = optimoptions('linprog','Algorithm','dual-simplex');
```

Set the initial point to `[]`, because the `dual-simplex` algorithm does not accept an initial point `x0`.

```
x0 = [];
```

Solve the linear program using the `'dual-simplex'` algorithm.

```
x = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)
```

```
Optimal solution found.


x =

    0.1875
    1.2500
```

Return the Objective Function Value

Calculate the solution and objective function value for a simple linear program.

The inequality constraints are

$$x(1) + x(2) \le 2$$

$$x(1) + x(2)/4 \le 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
A = [1 1
     1 1/4
     1 -1
     -1/4 -1
     -1 -1
     -1 1];

b = [2 1 2 1 -1 2];
```

The objective function is $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Solve the problem and return the objective function value.

```
[x,fval] = linprog(f,A,b)

Optimization terminated.

x =

    0.6667
    1.3333


fval =

   -1.1111
```

Obtain More Output to Examine the Solution Process

Obtain the exit flag and output structure to better understand the solution process and quality.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \le 2$$

$$x(1) + x(2)/4 \le 1$$

$$x(1) - x(2) \le 2$$

$$-x(1)/4 - x(2) \le 1$$

$$-x(1) - x(2) \le -1$$

$$-x(1) + x(2) \le 2.$$

```
A = [1  1
     1  1/4
     1  -1
     -1/4  -1
     -1  -1
     -1  1];

b = [2  1  2  1  -1  2];
```

Use the linear equality constraint $x(1) + x(2)/4 = 1/2$.

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

Set these bounds:

$$-1 \le x(1) \le 1.5$$

$$-0.5 \le x(2) \le 1.25.$$

```
lb = [-1,-0.5];
ub = [1.5,1.25];
```

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1  -1/3];
```

Set options to use the `'dual-simplex'` algorithm.

```
options = optimoptions('linprog','Algorithm','dual-simplex');
```

Set the initial point to `[]`, because the `dual-simplex` algorithm does not accept an initial point `x0`.

```
x0 = [];
```

Solve the linear program and request the function value, exit flag, and output structure.

```
[x,fval,exitflag,output] = linprog(f,A,b,Aeq,beq,lb,ub,x0,options)
```

```
Optimal solution found.


x =

    0.1875
    1.2500


fval =

   -0.6042


exitflag =

     1


output =

         iterations: 0
     constrviolation: 0
             message: 'Optimal solution found.'
           algorithm: 'dual-simplex'
       firstorderopt: 0
```

- `fval`, the objective function value, is larger than "Return the Objective Function Value", because there are more constraints.
- `exitflag` = 1 indicates that the solution is reliable.

- `output.iterations` = 0 indicates that `linprog` found the solution during presolve, and did not have to iterate at all.

Obtain Solution and Lagrange Multipliers

Solve a simple linear program and examine the solution and the Lagrange multipliers.

Use the objective function

$$f(x) = -5x_1 - 4x_2 - 6x_3.$$

```
f = [-5; -4; -6];
```

Use the linear inequality constraints

$$x_1 - x_2 + x_3 \leq 20$$

$$3x_1 + 2x_2 + 4x_3 \leq 42$$

$$3x_1 + 2x_2 \leq 30.$$

```
A =  [1 -1  1
      3  2  4
      3  2  0];
b = [20;42;30];
```

Constrain all variables to be positive:

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$x_3 \geq 0.$$

```
lb = zeros(3,1);
```

Set `Aeq` and `beq` to `[]`, indicating that there are no linear equality constraints.

```
Aeq = [];
beq = [];
```

Call `linprog`, obtaining the Lagrange multipliers.

```
[x,fval,exitflag,output,lambda] = linprog(f,A,b,Aeq,beq,lb);
```

```
Optimization terminated.
```

Examine the solution and Lagrange multipliers.

```
x,lambda.ineqlin,lambda.lower
```

```
x =

    0.0000
   15.0000
    3.0000


ans =

    0.0000
    1.5000
    0.5000


ans =

    1.0000
    0.0000
    0.0000
```

`lambda.ineqlin` is nonzero for the second and third components of `x`. This indicates that the second and third linear inequality constraints are satisfied with equalities:

$$3x_1 + 2x_2 + 4x_3 = 42$$

$$3x_1 + 2x_2 = 30.$$

Check that this is true:

```
A*x
```

```
ans =

  -12.0000
   42.0000
   30.0000
```

`lambda.lower` is nonzero for the first component of `x`. This indicates that `x(1)` is at its lower bound of 0.

- "Set Up a Linear Program" on page 1-12
- "Typical Linear Programming Problem" on page 8-22
- Maximize Long-Term Investments Using Linear Programming

Input Arguments

`f` — Objective function
real vector

Objective function, specified as a real vector. `linprog` attempts to find an `x` that minimizes the objective function
`f'*x = f(1)*x(1) + ... + f(N)*x(N),`

where `N` is the number of variables in the problem.

Example: `f = [1,3,5,-6]`

Data Types: `double`

`A` — Linear inequality constraints
real matrix

Linear inequality constraints, specified as a real matrix. `A` is an M-by-N matrix, where M is the number of inequalities, and `N` is the number of variables (number of elements in `x0`). For large problems, pass `A` as a sparse matrix.

`A` encodes the M linear inequalities
`A*x <= b,`

where `x` is the column vector of `N` variables `x(:)`, and `b` is a column vector with M elements.

For example, to specify
$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30,$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x-components add up to 1 or less, take A = ones(1,N) and b = 1

Data Types: double

b — Linear inequality constraints
real vector

Linear inequality constraints, specified as a real vector. b is an M-element vector related to the A matrix. If you pass b as a row vector, solvers internally convert b to the column vector b(:). For large problems, pass b as a sparse vector.

b encodes the M linear inequalities
A*x <= b,

where x is the column vector of N variables x(:), and A is a matrix of size M-by-N.

For example, to specify
$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30,$

give these constraints:

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x-components sum to 1 or less, take A = ones(1,N) and b = 1

Data Types: double

Aeq — Linear equality constraints
real matrix

Linear equality constraints, specified as a real matrix. Aeq is an Me-by-N matrix, where Me is the number of equalities, and N is the number of variables (number of elements in x0). For large problems, pass Aeq as a sparse matrix.

Aeq encodes the Me linear equalities
Aeq*x = beq,

where `x` is the column vector of `N` variables `x(:)`, and `beq` is a column vector with `Me` elements.

For example, to specify
$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20$,

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x-components sum to 1, take `Aeq = ones(1,N)` and `beq = 1`

Data Types: `double`

beq — Linear equality constraints
real vector

Linear equality constraints, specified as a real vector. `beq` is an `Me`-element vector related to the `Aeq` matrix. If you pass `beq` as a row vector, solvers internally convert `beq` to the column vector `beq(:)`. For large problems, pass `beq` as a sparse vector.

`beq` encodes the `Me` linear equalities
`Aeq*x = beq`,

where `x` is the column vector of `N` variables `x(:)`, and `Aeq` is a matrix of size `Meq`-by-`N`.

For example, to specify
$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20$,

give these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x-components sum to 1, take `Aeq = ones(1,N)` and `beq = 1`

Data Types: `double`

lb — Lower bounds
real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in x0 is equal to that of lb, then lb specifies that
x(i) >= lb(i) for all i.

If numel(lb) < numel(x0), then lb specifies that
x(i) >= lb(i) for 1 <= i <= numel(lb).

In this case, solvers issue a warning.

Example: To specify that all x-components are positive, lb = zeros(size(x0))

Data Types: double

ub — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in x0 is equal to that of ub, then ub specifies that
x(i) <= ub(i) for all i.

If numel(ub) < numel(x0), then ub specifies that
x(i) <= ub(i) for 1 <= i <= numel(ub).

In this case, solvers issue a warning.

Example: To specify that all x-components are less than one, ub = ones(size(x0))

Data Types: double

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Only the 'active-set' algorithm uses x0, all other solvers ignore its value.

Example: x0 = ones(size(f))

Data Types: double

options — Optimization options
output of optimoptions | structure as optimset returns

Optimization options, specified as the output of optimoptions or a structure as optimset returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose the optimization algorithm:

- `'interior-point-legacy'` (default)
- `'interior-point'`
- `'dual-simplex'`
- `'active-set'`
- `'simplex'`

For information on choosing the algorithm, see "Linear Programming Algorithms" on page 2-10.

Diagnostics	Display diagnostic information about the function to be minimized or solved. Choose `'off'` (default) or `'on'`.
Display	Level of display (see "Iterative Display" on page 3-16):

- `'final'` (default) displays just the final output.
- `'off'` or `'none'` displays no output.
- `'iter'` displays output at each iteration. The `'iter'` option is unavailable for the `'active-set'` algorithm.

LargeScale Use Algorithm instead	Use the `'interior-point-legacy'` algorithm when set to `'on'` (default). Use a medium-scale algorithm when set to `'off'` (see Simplex in options). For more information, see "Choosing the Algorithm" on page 2-7.
MaxIter	Maximum number of iterations allowed, a positive integer. The default is:

- 85 for the `'interior-point-legacy'` algorithm
- 200 for the `'interior-point'` algorithm
- 10*(numberOfEqualities + numberOfInequalities + numberOfVariables) for the `'dual-simplex'` algorithm
- 10*numberOfVariables for the `'simplex'` algorithm

- 10*max(numberOfVariables, numberOfInequalities + numberOfBounds) for the 'active-set' algorithm

See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.

TolFun	Termination tolerance on the function value, a positive scalar. The default is:

- 1e-8 for the 'interior-point' algorithm
- 1e-7 for the 'dual-simplex' algorithm
- 1e-6 for the 'interior-point' and 'simplex' algorithms
- The option is not used for the 'active-set' algorithm

TolFun measures dual feasibility tolerance.

interior-point Algorithm

Preprocess	Level of LP preprocessing prior to algorithm iterations. Specify 'basic' (default) or 'none'.
TolCon	Feasibility tolerance for constraints, a scalar from 1e-10 through 1e-3. TolCon measures primal feasibility tolerance. The default is 1e-6.

Dual-Simplex Algorithm

MaxTime	Maximum amount of time in seconds that the algorithm runs. The default is Inf.
Preprocess	Level of LP preprocessing prior to dual simplex algorithm iterations. Specify 'basic' (default) or 'none'.
TolCon	Feasibility tolerance for constraints, a scalar from 1e-10 through 1e-3. TolCon measures primal feasibility tolerance. The default is 1e-4.

Simplex Algorithm

Simplex Use Algorithm instead	If 'on', and if LargeScale is 'off', 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', meaning linprog uses an active-set algorithm.

Example: `options = optimoptions('linprog','Algorithm','interior-point','Display','iter')`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
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 created with `optimoptions`

You must supply at least the `solver` field in the `problem` structure.

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

Output Arguments

`x` — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `f` or `x0`.

`fval` — Objective function value at the solution
real number

Objective function value at the solution, returned as a real number. Generally, `fval = f'*x`.

exitflag — **Reason linprog stopped**
integer

Reason `linprog` stopped, returned as an integer.

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.

output — **Information about the optimization process**
structure

Information about the optimization process, returned as a structure with these fields.

iterations	Number of iterations
algorithm	Optimization algorithm used
cgiterations	0 (interior-point algorithm only, included for backward compatibility)
message	Exit message
constrviolation	Maximum of constraint functions
firstorderopt	First-order optimality measure

lambda — **Lagrange multipliers at the solution**
structure

Lagrange multipliers at the solution, returned as a structure with these fields.

lower	Lower bounds corresponding to `lb`
upper	Upper bounds corresponding to `ub`

`ineqlin`	Linear inequalities corresponding to `A` and `b`
`eqlin`	Linear equalities corresponding to `Aeq` and `beq`

Limitations

- The `'active-set'` algorithm has only `'off'` and `'final'` as `Display` options; you cannot choose the iterative display option, `'iter'`.

More About

Algorithms

Interior-Point-Legacy Algorithm

The `'interior-point-legacy'` 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 "Interior-Point-Legacy Linear Programming" on page 8-8.

The first stage of the algorithm might involve some preprocessing of the constraints (see "Interior-Point-Legacy Linear Programming" on page 8-8). Several conditions might cause `linprog` to exit with an infeasibility message. In each case, `linprog` returns a negative `exitflag`, indicating to indicate failure.

- If a row of all zeros is detected in `Aeq`, but the corresponding element of `beq` is not zero, then 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, then 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, then 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, then 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, then 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 can cause such a row to occur.

When the preprocessing finishes, the iterative part of the algorithm begins until the stopping criteria are met. (For more information about residuals, the primal problem, the dual problem, and the related stopping criteria, see "Interior-Point-Legacy Linear Programming" on page 8-8.) 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 messages indicating that the dual, the primal, or both appear to be infeasible.

- `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.

For example, the primal (objective) can be unbounded and the primal residual, which is a measure of primal constraint satisfaction, can be small.

Interior-Point Algorithm

The `'interior-point'` algorithm is similar to `'interior-point-legacy'`, but with a more efficient factorization routine, and with different preprocessing. See "Interior-Point `linprog` Algorithm" on page 8-2.

Dual-Simplex Algorithm

For a description, see "Dual-Simplex Algorithm" on page 8-19.

Active-Set and Simplex Algorithms

The `'active-set'` algorithm 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. For details, see "Active-Set `linprog` Algorithm" on page 8-11.

For a description of the `'simplex'` algorithm, see "`linprog` Simplex Algorithm" on page 8-15.

`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.

- "Optimization Problem Setup"
- "Linear Programming Algorithms" on page 8-2

References

[1] Dantzig, G.B., A. Orden, and P. Wolfe. "Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints." *Pacific Journal Math.,* Vol. 5, 1955, pp. 183–195.

[2] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization*, Vol. 2, 1992, pp. 575–601.

[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

intlinprog | mpsread | optimoptions | quadprog

Introduced before R2006a

lsqcurvefit

Solve nonlinear curve-fitting (data-fitting) problems in least-squares sense

Nonlinear least-squares solver

Find coefficients x that solve the problem

$$\min_{x} \|F(x,xdata) - ydata\|_2^2 = \min_{x} \sum_{i} \left(F(x, xdata_i) - ydata_i \right)^2,$$

given input data *xdata*, and the observed output *ydata*, where *xdata* and *ydata* are matrices or vectors, and $F(x, xdata)$ is a matrix-valued or vector-valued function of the same size as *ydata*.

Optionally, the components of x can have lower and upper bounds *lb*, and *ub*. The arguments x, *lb*, and *ub* can be vectors or matrices; see "Matrix Arguments" on page 2-30.

The `lsqcurvefit` function uses the same algorithm as `lsqnonlin`. `lsqcurvefit` simply provides a convenient interface for data-fitting problems.

Rather than compute the sum of squares, `lsqcurvefit` requires the user-defined function to compute the *vector*-valued function

$$F(x,xdata) = \begin{bmatrix} F(x,xdata(1)) \\ F(x,xdata(2)) \\ \vdots \\ F(x,xdata(k)) \end{bmatrix}.$$

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,exitflag,output] = lsqcurvefit( ___ )
```

```
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
lsqcurvefit( ___ )
```

Description

`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`.

Note: "Passing Extra Parameters" on page 2-50 explains how to pass extra parameters to the vector function `fun(x)`, if necessary.

`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 $lb \leq x \leq ub$. You can fix the solution component `x(i)` by specifying `lb(i) = ub(i)`.

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 $lb \leq x \leq ub$ are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

`x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)` minimizes with the optimization options specified in `options`. Use `optimoptions` 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 14-208. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[x,resnorm] = lsqcurvefit( ___ )`, for any input arguments, returns the value of the squared 2-norm of the residual at `x`: `sum((fun(x,xdata)-ydata).^2)`.

`[x,resnorm,residual,exitflag,output] = lsqcurvefit( ___ )` additionally returns the value of the residual `fun(x,xdata)-ydata` at the solution `x`, a value

exitflag that describes the exit condition, and a structure `output` that contains information about the optimization process.

```
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
lsqcurvefit( ___ )
```
additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution x, and the Jacobian of `fun` at the solution x.

Examples

Simple Exponential Fit

Suppose that you have observation time data xdata and observed response data ydata, and you want to find parameters $x(1)$ and $x(2)$ to fit a model of the form

$$\text{ydata} = x(1)\exp(x(2)\text{xdata}).$$

Input the observation times and responses.

```
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];
```

Create a simple exponential decay model.

```
fun = @(x,xdata)x(1)*exp(x(2)*xdata);
```

Fit the model using the starting point x0 = [100,-1].

```
x0 = [100,-1];
x = lsqcurvefit(fun,x0,xdata,ydata)
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.



x =
```

```
498.8309    -0.1013
```

Plot the data and the fitted curve.

```
times = linspace(xdata(1),xdata(end));
plot(xdata,ydata,'ko',times,fun(x,times),'b-')
legend('Data','Fitted exponential')
title('Data and Fitted Curve')
```
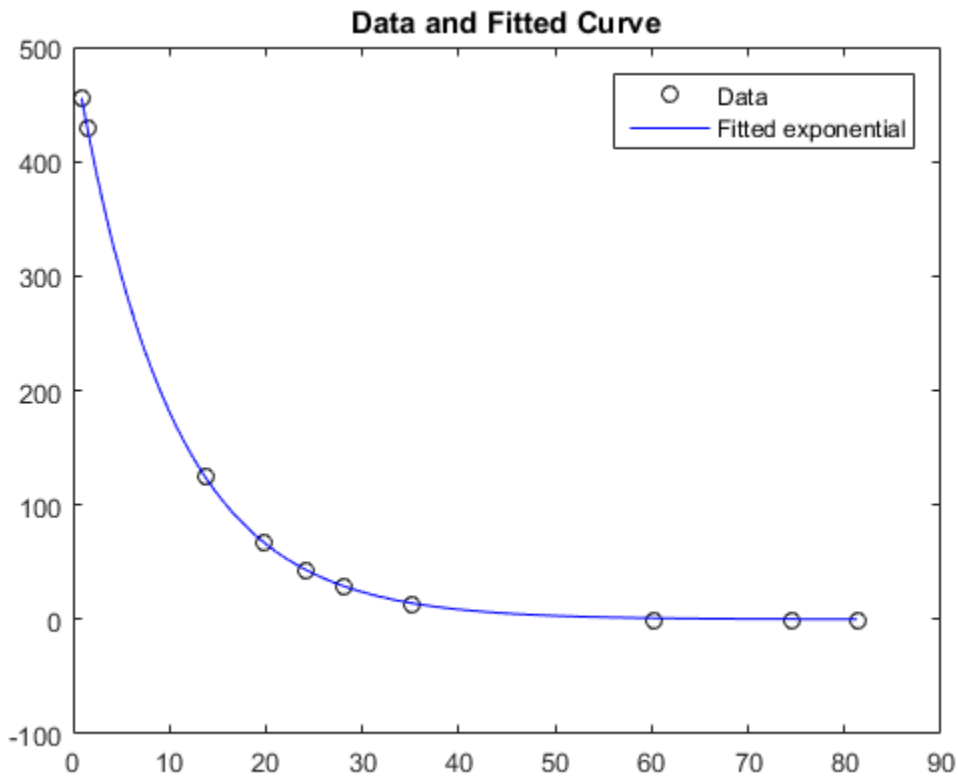

Best Fit with Bound Constraints

Find the best exponential fit to data where the fitting parameters are constrained.

Generate data from an exponential decay model plus noise. The model is

$$y = \exp(-1.3t) + \varepsilon,$$

with t ranging from 0 through 3, and ε normally distributed noise with mean 0 and standard deviation 0.05.

```
rng default % for reproducibility
xdata = linspace(0,3);
ydata = exp(-1.3*xdata) + 0.05*randn(size(xdata));
```

The problem is: given the data (`xdata`, `ydata`), find the exponential decay model $y = x(1)\exp(x(2)\text{xdata})$ that best fits the data, with the parameters bounded as follows:

$$0 \le x(1) \le 3/4$$

$$-2 \le x(2) \le -1.$$

```
lb = [0,-2];
ub = [3/4,-1];
```

Create the model.

```
fun = @(x,xdata)x(1)*exp(x(2)*xdata);
```

Create an initial guess.

```
x0 = [1/2,-2];
```

Solve the bounded fitting problem.

```
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub)
```

```
Local minimum found.

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

```
x =
```

```
0.7500    -1.0000
```

Examine how well the resulting curve fits the data. Because the bounds keep the solution away from the true values, the fit is mediocre.

```
plot(xdata,ydata,'ko',xdata,fun(x,xdata),'b-')
legend('Data','Fitted exponential')
title('Data and Fitted Curve')
```
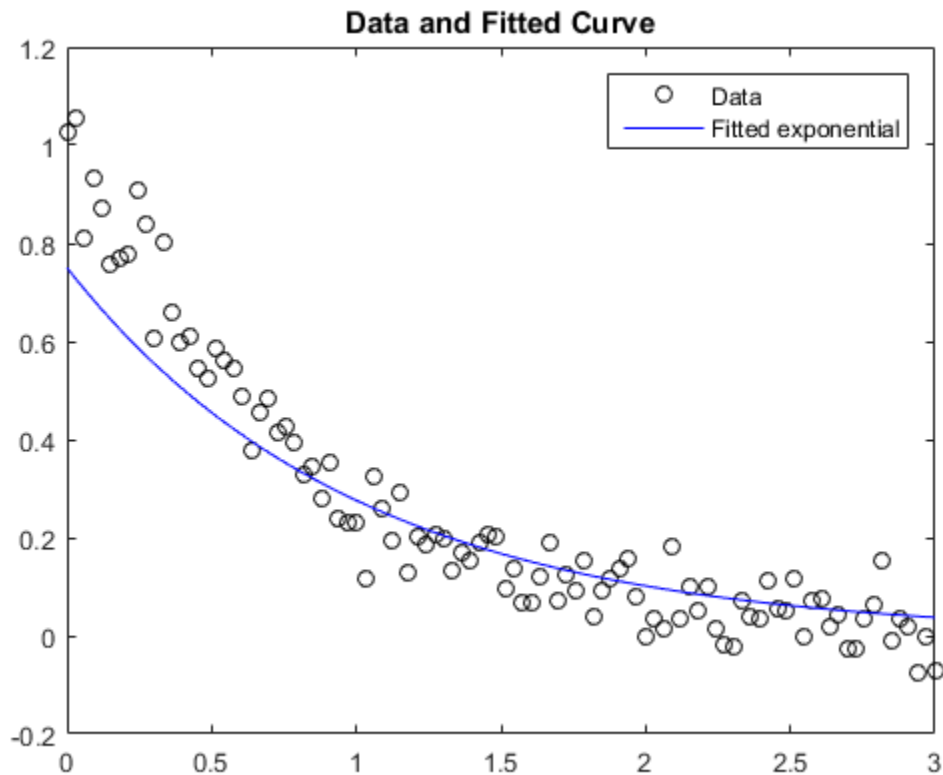

Compare Algorithms

Compare the results of fitting with the default `'trust-region-reflective'` algorithm and the `'levenberg-marquardt'` algorithm.

Suppose that you have observation time data `xdata` and observed response data `ydata`, and you want to find parameters $x(1)$ and $x(2)$ to fit a model of the form

$$\text{ydata} = x(1)\exp\left(x(2)\text{xdata}\right).$$

Input the observation times and responses.

```
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];
```

Create a simple exponential decay model.

```
fun = @(x,xdata)x(1)*exp(x(2)*xdata);
```

Fit the model using the starting point `x0 = [100,-1]`.

```
x0 = [100,-1];
x = lsqcurvefit(fun,x0,xdata,ydata)
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

```
x =

  498.8309   -0.1013
```

Compare the solution with that of a `'levenberg-marquardt'` fit.

```
options = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
lb = [];
ub = [];
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)
```

```
Local minimum possible.

lsqcurvefit stopped because the relative size of the current step is less than
the default value of the step size tolerance.
```

```
x =

   498.8309   -0.1013
```

The two algorithms converged to the same solution. Plot the data and the fitted exponential model.

```
times = linspace(xdata(1),xdata(end));
plot(xdata,ydata,'ko',times,fun(x,times),'b-')
legend('Data','Fitted exponential')
title('Data and Fitted Curve')
```
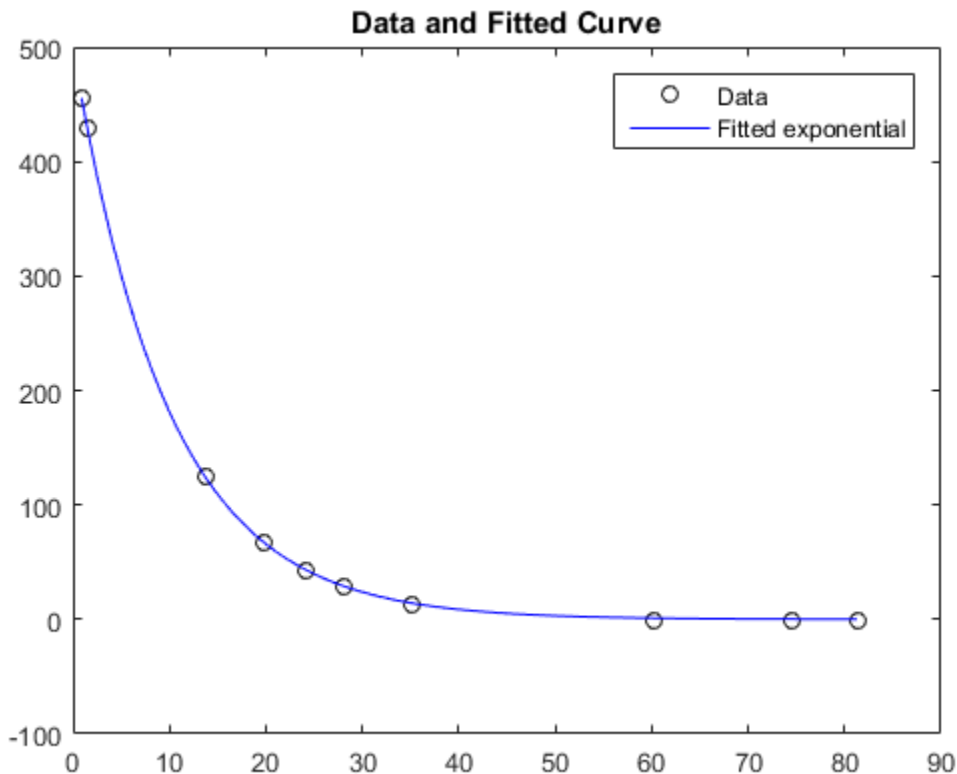
Compare Algorithms and Examine Solution Process

Compare the results of fitting with the default `'trust-region-reflective'` algorithm and the `'levenberg-marquardt'` algorithm. Examine the solution process to see which is more efficient in this case.

Suppose that you have observation time data xdata and observed response data ydata, and you want to find parameters $x(1)$ and $x(2)$ to fit a model of the form

$$\text{ydata} = x(1) \exp{(x(2)\text{xdata})}.$$

Input the observation times and responses.

```
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];
```

Create a simple exponential decay model.

```
fun = @(x,xdata)x(1)*exp(x(2)*xdata);
```

Fit the model using the starting point `x0 = [100,-1]`.

```
x0 = [100,-1];
[x,resnorm,residual,exitflag,output] = lsqcurvefit(fun,x0,xdata,ydata);
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

Compare the solution with that of a `'levenberg-marquardt'` fit.

```
options = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
lb = [];
ub = [];
[x2,resnorm2,residual2,exitflag2,output2] = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,option
```

```
Local minimum possible.

lsqcurvefit stopped because the relative size of the current step is less than
the default value of the step size tolerance.
```

Are the solutions equivalent?

```
norm(x-x2)
```

```
ans =

    2.0619e-06
```

Yes, the solutions are equivalent.

Which algorithm took fewer function evaluations to arrive at the solution?

```
fprintf(['The ''trust-region-reflective'' algorithm took %d function evaluations,\n',...
    'and the ''levenberg-marquardt'' algorithm took %d function evaluations.\n'],...
    output.funcCount,output2.funcCount)

The 'trust-region-reflective' algorithm took 87 function evaluations,
and the 'levenberg-marquardt' algorithm took 72 function evaluations.
```

Plot the data and the fitted exponential model.

```
times = linspace(xdata(1),xdata(end));
plot(xdata,ydata,'ko',times,fun(x,times),'b-')
legend('Data','Fitted exponential')
title('Data and Fitted Curve')
```

Data and Fitted Curve
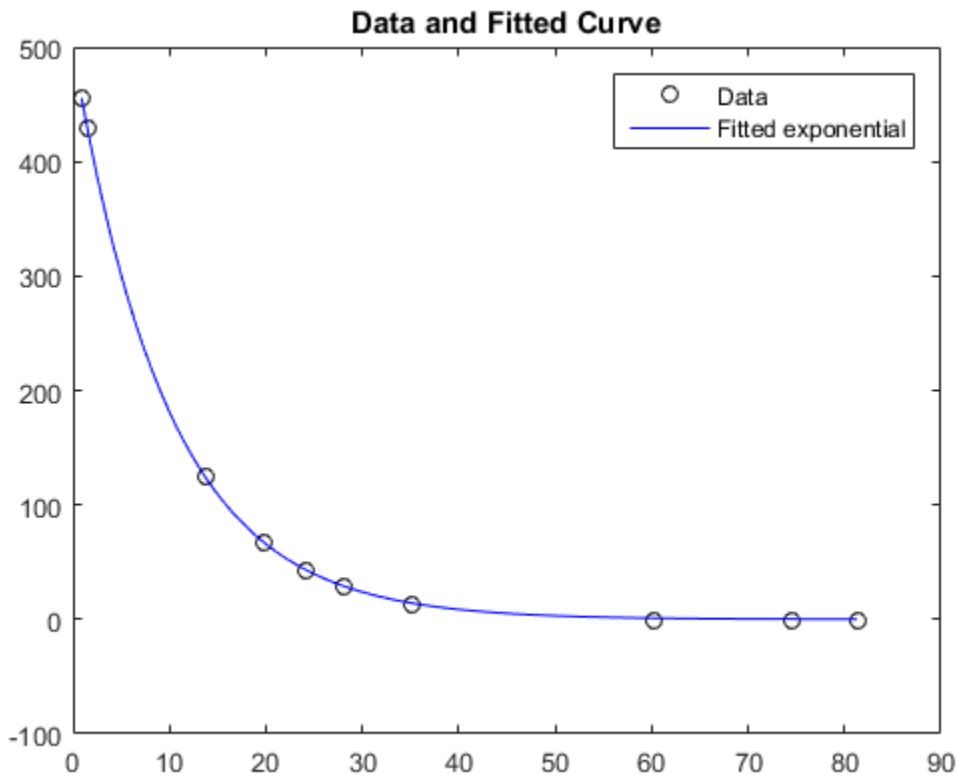
The fit looks good. How large are the residuals?

```
fprintf(['The ''trust-region-reflective'' algorithm has residual norm %f,\n',...
    'and the ''levenberg-marquardt'' algorithm has residual norm %f.\n'],...
    resnorm,resnorm2)
```

```
The 'trust-region-reflective' algorithm has residual norm 9.504887,
and the 'levenberg-marquardt' algorithm has residual norm 9.504887.
```

· "Nonlinear Least Squares (Curve Fitting)"

Input Arguments

fun — Function you want to fit
function handle | name of function

Function you want to fit, specified as a function handle or the name of a function. `fun` is a function that takes two inputs: a vector or matrix `x`, and a vector or matrix `xdata`. `fun` returns a vector or matrix `F`, the objective function evaluated at `x` and `xdata`. The function fun can be specified as a function handle for a function file:

```
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, xdata
```

`fun` can also be a function handle for an anonymous function.

```
f = @(x,xdata)x(1)*xdata.^2+x(2)*sin(xdata);
x = lsqcurvefit(f,x0,xdata,ydata);
```

If the user-defined values for `x` and `F` are matrices, `lsqcurvefit` converts them to vectors using linear indexing.

Note `fun` should return `fun(x,xdata)`, and not the sum-of-squares `sum((fun(x,xdata)-ydata).^2)`. `lsqcurvefit` implicitly computes the sum of squares of the components of `fun(x,xdata)-ydata`. See "Examples" on page 14-   .

If the Jacobian can also be computed *and* the Jacobian option is `'on'`, set by

```
options = optimoptions('lsqcurvefit','Jacobian','on')
```

then the function `fun` must return a second output argument with 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`, 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 "Writing Vector and Matrix Objective Functions" on page 2-25.

Example: `@(x,xdata)x(1)*exp(-x(2)*xdata)`

Data Types: `char` | `function_handle`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

xdata — Input data for model
real vector | real array

Input data for model, specified as a real vector or real array. The model is `ydata = fun(x,xdata)`,

where `xdata` and `ydata` are fixed arrays, and `x` is the array of parameters that `lsqcurvefit` changes to search for a minimum sum of squares.

Example: `xdata = [1,2,3,4]`

Data Types: `double`

ydata — Response data for model
real vector | real array

Response data for model, specified as a real vector or real array. The model is `ydata = fun(x,xdata)`,

where `xdata` and `ydata` are fixed arrays, and `x` is the array of parameters that `lsqcurvefit` changes to search for a minimum sum of squares.

The `ydata` array must be the same size and shape as the array `fun(x0,xdata)`.

Example: `ydata = [1,2,3,4]`

Data Types: `double`

`lb` — Lower bounds
real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `lb`, then `lb` specifies that
`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(x0)`, then `lb` specifies that
`x(i) >= lb(i)` for `1 <= i <= numel(lb)`.

In this case, solvers issue a warning.

Example: To specify that all x-components are positive, `lb = zeros(size(x0))`

Data Types: `double`

`ub` — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `ub`, then `ub` specifies that
`x(i) <= ub(i)` for all `i`.

If `numel(ub) < numel(x0)`, then `ub` specifies that
`x(i) <= ub(i)` for `1 <= i <= numel(ub)`.

In this case, solvers issue a warning.

Example: To specify that all x-components are less than one, `ub = ones(size(x0))`

Data Types: `double`

`options` — Optimization options
output of `optimoptions` | structure such as `optimset` returns

Optimization options, specified as the output of `optimoptions` or a structure such as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose between `'trust-region-reflective'` (default) and `'levenberg-marquardt'`. Set the initial Levenberg-Marquardt parameter λ by setting `Algorithm` to a cell array such as `{'levenberg-marquardt',.005}`. The default $\lambda = 0.01$.
	The `Algorithm` option specifies a preference for which algorithm to use. It is only a preference, because certain conditions must be met to use each 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`. The Levenberg-Marquardt algorithm does not handle bound constraints. For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
DerivativeCheck	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are `'off'` (default) or `'on'`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. Choices are `'off'` (default) or `'on'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
Display	Level of display (see "Iterative Display" on page 3-16):

- `'off'` or `'none'` displays no output.
- `'iter'` displays output at each iteration, and gives the default exit message.
- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.

- `'final'` (default) displays just the final output, and gives the default exit message.
- `'final-detailed'` displays just the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are
	`delta = v.*sign′(x).*max(abs(x),TypicalX);`
	where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are
	`delta = v.*max(abs(x),TypicalX);`
	Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.

FinDiffType	Finite differences, used to estimate gradients, are either `'forward'` (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 could take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
FunValCheck	Check whether function values are valid. `'on'` displays an error when the function returns a value that is `complex`, `Inf`, or `NaN`. The default `'off'` displays no error.
Jacobian	If `'off'` (default), the solver approximates the Jacobian using finite differences. If `'on'`, the solver uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function.
MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default is `400`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.

| OutputFcn | Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20. |
| PlotFcns | Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`): |

- `@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 optimality measure.

For information on writing a custom plot function, see "Plot Functions" on page 13-29.

TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on `x`, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TypicalX	Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. The solver uses `TypicalX` for scaling finite differences for gradient estimation.

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. The solver uses `Jinfo` to compute the preconditioner. See "Passing Extra Parameters" on page 2-50 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'Jacobian'` must be set to `'on'` for the solver to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75 and "Jacobian Multiply Function with Linear Least Squares" on page 10-26 for similar examples.

JacobPattern	Sparsity pattern of the Jacobian for finite differencing. Set `JacobPattern(i,j) = 1` when `fun(i)` depends on `x(j)`. Otherwise, set `JacobPattern(i,j) = 0`. In other words, `JacobPattern(i,j) = 1` when you can have ∂`fun(i)`/∂`x(j)` $\neq 0$.
	Use `JacobPattern` when it is inconvenient to compute the Jacobian matrix `J` in `fun`, though you can determine (say, by inspection) when `fun(i)` depends on `x(j)`. The solver can approximate `J` via sparse finite differences when you give `JacobPattern`.
	If the structure is unknown, do not set `JacobPattern`. The default behavior is as if `JacobPattern` is a dense matrix of ones. Then the solver computes a full finite-difference approximation in each iteration. This can be expensive for large problems, so it is usually better to determine the sparsity structure.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,numberOfVariables/2)`. For more information, see "Large Scale Nonlinear Least Squares" on page 10-5.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG, a nonnegative integer. 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, a positive scalar. The default is `0.1`.

Levenberg-Marquardt Algorithm

InitDamping	Initial value of the Levenberg-Marquardt parameter, a positive scalar. Default is `1e-2`. For details, see "Levenberg-Marquardt Method" on page 10-7.

ScaleProblem	'Jacobian' can sometimes improve the convergence of a poorly scaled problem; the default is 'none'.

Example: options = optimoptions('lsqcurvefit','Jacobian','on')

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
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 created with optimoptions

You must supply at least the objective, x0, solver, xdata, ydata, and options fields in the problem structure.

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: struct

Output Arguments

x — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when exitflag is positive. For information on the quality of the solution, see "When the Solver Succeeds" on page 4-22.

resnorm — Norm of the residual
nonnegative real

Norm of the residual, returned as a nonnegative real. `resnorm` is the squared 2-norm of the residual at `x`: `sum((fun(x,xdata)-ydata).^2)`.

residual — Value of objective function at solution
vector

Value of objective function at solution, returned as a vector. In general, `residual = fun(x,xdata)-ydata`.

exitflag — Reason the solver stopped
integer

Reason the solver stopped, returned as an integer.

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.MaxFunEvals`.
-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.

output — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

firstorderopt	Measure of first-order optimality

iterations	Number of iterations taken
funcCount	The number of function evaluations
cgiterations	Total number of PCG iterations (trust-region-reflective algorithm only)
stepsize	Final displacement in x
algorithm	Optimization algorithm used
message	Exit message

`lambda` — Lagrange multipliers at the solution
structure

Lagrange multipliers at the solution, returned as a structure with fields:

lower	Lower bounds lb
upper	Upper bounds ub

`jacobian` — Jacobian at the solution
real matrix

Jacobian at the solution, returned as a real matrix. `jacobian(i,j)` is the partial derivative of `fun(i)` with respect to `x(j)` at the solution x.

Limitations

- The Levenberg-Marquardt algorithm does not handle bound constraints.

- The trust-region-reflective algorithm 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, `lsqcurvefit` uses the Levenberg-Marquardt algorithm.

   Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems that have both of these characteristics cannot be solved by `lsqcurvefit`.

- `lsqcurvefit` can solve complex-valued problems directly with the `levenberg-marquardt` algorithm. However, this algorithm does not accept bound constraints. For a complex problem with bound constraints, split the variables into real and

imaginary parts, and use the `trust-region-reflective` algorithm. See "Fit a Model to Complex-Valued Data" on page 10-33.

- 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, `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.

You can use the trust-region reflective algorithm in `lsqnonlin`, `lsqcurvefit`, and `fsolve` with small- to medium-scale problems without computing the Jacobian in `fun` or providing the Jacobian sparsity pattern. (This also applies to 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 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. You can find out only by running it and seeing if MATLAB runs within the amount of virtual memory available on your system.

More About

Algorithms

The Levenberg-Marquardt and trust-region-reflective methods are based on the nonlinear least-squares algorithms also used in `fsolve`.

- The default 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 Least Squares" on page 10-3.

- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 10-7.

- "Optimization Problem Setup"
- "Least-Squares (Model Fitting) Algorithms" on page 10-2

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, 1996, pp. 418–445.

[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, 1994, pp. 189–224.

[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*, 1944, pp. 164–168.

[5] Marquardt, D. "An Algorithm for Least-squares Estimation of Nonlinear Parameters." *SIAM Journal Applied Mathematics*, Vol. 11, 1963, pp. 431–441.

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

[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
`fsolve` | `lsqnonlin` | `optimoptions`

Introduced before R2006a

lsqlin

Solve constrained linear least-squares problems

Linear least-squares solver with bounds or linear constraints.

Solves least-squares curve fitting problems of the form

$$\min_{x} \frac{1}{2} \| C \cdot x - d \|_2^2 \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

Syntax

```
x = lsqlin(C,d,A,b)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)
x = lsqlin(problem)
[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`.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)` adds linear equality constraints `Aeq*x = beq` and bounds `lb ≤ x ≤ ub`. If you do not need certain constraints such as `Aeq` and `beq`, set them to `[]`. If `x(i)` is unbounded below, set `lb(i) = -Inf`, and if `x(i)` is unbounded above, set `ub(i) = Inf`.

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with an initial point `x0` and the optimization options specified in `options`. Use `optimoptions` to set these options. If you do not want to include an initial point, set the `x0` argument to `[]`.

`x = lsqlin(problem)` finds the minimum for `problem`, where `problem` is a structure. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

[x,resnorm,residual,exitflag,output,lambda] = lsqlin(___), for any input arguments described above, returns:

- The squared 2-norm of the residual resnorm $= \left\| C \cdot x - d \right\|_2^2$

- The residual residual = C*x - d
- A value exitflag describing the exit condition
- A structure output containing information about the optimization process
- A structure lambda containing the Lagrange multipliers

The factor ½ in the definition of the problem affects the values in the lambda structure.

Examples

Least Squares with Linear Inequality Constraints

Find the x that minimizes the norm of C*x - d for an overdetermined problem with linear inequality constraints.

Specify the problem and constraints.

```
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];
```

Call lsqlin to solve the problem.

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

```
Warning: The trust-region-reflective algorithm can handle bound constraints
only;
    using active-set algorithm instead.
Optimization terminated.

x =

    0.1299
   -0.5757
    0.4251
    0.2438
```

Least Squares with Linear Constraints and Bounds

Find the x that minimizes the norm of C*x - d for an overdetermined problem with linear inequality constraints and bounds.

Specify the problem and constraints.

```
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];
Aeq = [3 5 7 9];
beq = 4;
lb = -0.1*ones(4,1);
ub = 2*ones(4,1);
```

Call lsqlin to solve the problem.

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

```
Warning: The trust-region-reflective algorithm can handle bound constraints
only;
```

```
     using active-set algorithm instead.
Optimization terminated.

x =

   -0.1000
   -0.1000
    0.1599
    0.4090
```

Least Squares with Start Point and Nondefault Options

Find the x that minimizes the norm of `C*x - d` for an overdetermined problem with linear inequality constraints. Use a start point and nondefault options.

Specify the problem and constraints.

```
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];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Set a start point.

```
x0 = 0.1*ones(4,1);
```

Set options to choose the `'active-set'` algorithm, which is the only algorithm that uses a start point.

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

Call `lsqlin` to solve the problem.

```
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)
```

```
Optimization terminated.

x =

    0.1299
   -0.5757
    0.4251
    0.2438
```

Return All Outputs

Obtain and interpret all `lsqlin` outputs.

Define a problem with linear inequality constraints and bounds. The problem is overdetermined because there are four columns in the `C` matrix but five rows. This means the problem has four unknowns and five conditions, even before including the linear constraints and 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);
```

Set options to use the `'interior-point'` algorithm.

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

The `'interior-point'` algorithm does not use an initial point, so set `x0` to `[]`.

```
x0 = [];
```

Call `lsqlin` with all outputs.

```
[x,resnorm,residual,exitflag,output,lambda] = ...
    lsqlin(C,d,A,b,[],[],lb,ub,x0,options)
```

```
Minimum found that satisfies the constraints.

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


x =

   -0.1000
   -0.1000
    0.2152
    0.3502


resnorm =

    0.1672


residual =

    0.0455
    0.0764
   -0.3562
    0.1620
    0.0784


exitflag =
```

```
     1


output =

           message: 'Minimum found that satisfies the constraints....'
         algorithm: 'interior-point'
      firstorderopt: 1.6296e-08
    constrviolation: 0
         iterations: 6
       cgiterations: []


lambda =

    ineqlin: [3x1 double]
      eqlin: [0x1 double]
      lower: [4x1 double]
      upper: [4x1 double]
```

Examine the nonzero Lagrange multiplier fields in more detail. First examine the Lagrange multipliers for the linear inequality constraint.

```
lambda.ineqlin


ans =

    0.0000
    0.2392
    0.0000
```

Lagrange multipliers are nonzero exactly when the solution is on the corresponding constraint boundary. In other words, Lagrange multipliers are nonzero when the corresponding constraint is active. `lambda.ineqlin(2)` is nonzero. This means that the second element in `A*x` should equal the second element in `b`, because the constraint is active.

```
[A(2,:)*x,b(2)]


ans =
```

```
    0.2026    0.2026
```

Now examine the Lagrange multipliers for the lower and upper bound constraints.

```
lambda.lower
```

```
ans =

    0.0409
    0.2784
    0.0000
    0.0000
```

```
lambda.upper
```

```
ans =

   1.0e-10 *

    0.4665
    0.4751
    0.5537
    0.6247
```

The first two elements of `lambda.lower` are nonzero. You see that `x(1)` and `x(2)` are at their lower bounds, `-0.1`. All elements of `lambda.upper` are essentially zero, and you see that all components of `x` are less than their upper bound, `2`.

- "Linear Least Squares with Bound Constraints" on page 10-21
- "Optimization App with the lsqlin Solver" on page 10-23
- "Jacobian Multiply Function with Linear Least Squares" on page 10-26

Input Arguments

C — Multiplier matrix
real matrix

Multiplier matrix, specified as a matrix of doubles. `C` represents the multiplier of the solution `x` in the expression `C*x - d`. `C` is M-by-N, where `M` is the number of equations, and `N` is the number of elements of `x`.

Example: `C = [1,4;2,5;7,8]`

Data Types: `double`

d — Constant vector
real vector

Constant vector, specified as a vector of doubles. `d` represents the additive constant term in the expression `C*x - d`. `d` is M-by-1, where `M` is the number of equations.

Example: `d = [5;0;-12]`

Data Types: `double`

A — Linear inequality constraint matrix
real matrix

Linear inequality constraint matrix, specified as a matrix of doubles. `A` represents the linear coefficients in the constraints `A*x ≤ b`. `A` has size `Mineq`-by-N, where `Mineq` is the number of constraints and `N` is the number of elements of `x`. To save memory, pass `A` as a sparse matrix.

Example: `A = [4,3;2,0;4,-1];` means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: `double`

b — Linear inequality constraint vector
real vector

Linear inequality constraint vector, specified as a vector of doubles. `b` represents the constant vector in the constraints `A*x ≤ b`. `b` has length `Mineq`, where `A` is `Mineq`-by-N.

Example: `[4,0]`

Data Types: `double`

Aeq — Linear equality constraint matrix
`[]` (default) | real matrix

Linear equality constraint matrix, specified as a matrix of doubles. `Aeq` represents the linear coefficients in the constraints `Aeq*x = beq`. `Aeq` has size `Meq`-by-N, where `Meq` is

the number of constraints and N is the number of elements of x. To save memory, pass Aeq as a sparse matrix.

Example: A = [4,3;2,0;4,-1]; means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: `double`

beq — Linear equality constraint vector
[] (default) | real vector

Linear equality constraint vector, specified as a vector of doubles. beq represents the constant vector in the constraints Aeq*x = beq. beq has length Meq, where Aeq is Meq-by-N.

Example: [4,0]

Data Types: `double`

lb — Lower bounds
[] (default) | real vector or array

Lower bounds, specified as a vector or array of doubles. lb represents the lower bounds elementwise in lb $\le$ x $\le$ ub.

Internally, lsqlin converts an array lb to the vector lb(:).

Example: lb = [0;-Inf;4] means x(1) $\ge$ 0, x(3) $\ge$ 4.

Data Types: `double`

ub — Upper bounds
[] (default) | real vector or array

Upper bounds, specified as a vector or array of doubles. ub represents the upper bounds elementwise in lb $\le$ x $\le$ ub.

Internally, lsqlin converts an array ub to the vector ub(:).

Example: ub = [Inf;4;10] means x(2) $\le$ 4, x(3) $\le$ 10.

Data Types: `double`

x0 — Initial point
[] (default) | real vector or array

Initial point for the solution process, specified as a vector or array of doubles. x0 is used only by the 'active-set' algorithm. Optional.

If you do not provide an x0 for the 'active-set' algorithm, lsqlin sets x0 to the zero vector. If any component of this zero vector x0 violates the bounds, lsqlin sets x0 to a point in the interior of the box defined by the bounds.

Example: x0 = [4;-3]

Data Types: double

options — Options for `lsqlin`
options created using optimoptions or the Optimization app

Options for lsqlin, specified as the output of the optimoptions function or the Optimization app.

All Algorithms

Algorithm	Choose the algorithm:
	• 'trust-region-reflective' (default)
	• 'interior-point'
	• 'active-set'
	The 'trust-region-reflective' algorithm allows *only* upper and lower bounds, meaning no linear inequalities or equalities. If you specify both the 'trust-region-reflective' and linear constraints, lsqlin uses the active-set algorithm.
	The trust-region-reflective algorithm does not allow equal upper and lower bounds. Use one of the other algorithms for this case.
	For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
Diagnostics	Display diagnostic information about the function to be minimized or solved. The choices are 'on' or the default 'off'.
Display	Level of display returned to the command line.
	• 'off' or 'none' displays no output.

- `'final'` displays just the final output (default).

The `'interior-point'` algorithm allows additional values:

- `'iter'` gives iterative display.
- `'iter-detailed'` gives iterative display with a detailed exit message.
- `'final-detailed'` displays just the final output, with a detailed exit message.

LargeScale Use Algorithm instead of LargeScale.	Use the `'trust-region-reflective'` algorithm if possible when set to `'on'` (default). Use the `'active-set'` algorithm when set to `'off'`. You cannot choose the `'interior-point'` algorithm using LargeScale.
MaxIter	Maximum number of iterations allowed, a positive integer. The default value is 200.

trust-region-reflective Algorithm Options

JacobMult	Function handle for the 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 2-50 for information on how to supply extra parameters if necessary.

	See "Jacobian Multiply Function with Linear Least Squares" on page 10-26 for an example.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,floor(numberOfVariables/2))`. For more information, see "Trust-Region-Reflective Algorithm" on page 14-237.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG (preconditioned conjugate gradient). 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 toward the solution. For more information, see "Trust-Region-Reflective Algorithm" on page 14-237.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `100*eps`, about `2.2204e-14`.
TolPCG	Termination tolerance on the PCG (preconditioned conjugate gradient) iteration, a positive scalar. The default is `0.1`.
TypicalX	Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `lsqlin` uses `TypicalX` internally for scaling. `TypicalX` has an effect only when `x` has unbounded components, and when a `TypicalX` value for an unbounded component is larger than `1`.

interior-point Algorithm Options

TolCon	Tolerance on the constraint violation, a positive scalar. The default is `1e-8`.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-8`.

problem — Optimization problem
structure

Optimization problem, specified as a structure with the following fields.

C	Matrix multiplier in the term `C*x - d`
d	Additive constant in the term `C*x - d`
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 created with `optimoptions`

Create the `problem` structure by exporting a problem from the Optimization app, as described in "Exporting Your Work" on page 5-11.

Data Types: `struct`

Output Arguments

`x` — Solution
real vector

Solution, returned as a vector that minimizes the norm of `C*x-d` subject to all bounds and linear constraints.

`resnorm` — Objective value
real scalar

Objective value, returned as the scalar value `norm(C*x-d)^2`.

`residual` — Solution residuals
real vector

Solution residuals, returned as the vector `C*x-d`.

`exitflag` — Algorithm stopping condition
integer

Algorithm stopping condition, returned as an integer identifying the reason the algorithm stopped. The following lists the values of `exitflag` and the corresponding reasons `lsqlin` stopped.

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.

The exit message for the `interior-point` algorithm can give more details on the reason `lsqlin` stopped, such as exceeding a tolerance. See "Exit Flags and Exit Messages" on page 3-3.

output — Solution process summary

structure

Solution process summary, returned as a structure containing information about the optimization process.

iterations	Number of iterations the solver took.
algorithm	One of these algorithms:
	• `'interior-point'`
	• `'active-set'`
	• `'trust-region-reflective'`
constrviolation	Constraint violation that is positive for violated constraints (not returned for the `'trust-region-reflective'` algorithm).
	`constrviolation = max([0;norm(Aeq*x-beq, inf);(lb-x);(x-ub);(A*x-b)])`
message	Exit message.

| firstorderopt | First-order optimality at the solution. See "First-Order Optimality Measure" on page 3-11. |
| cgiterations | Number of conjugate gradient iterations the solver performed. Nonempty only for the `'trust-region-reflective'` algorithm. |

See "Output Structures" on page 3-24.

`lambda` — Lagrange multipliers
structure

Lagrange multipliers, returned as a structure with the following fields.

lower	Lower bounds `lb`
upper	Upper bounds `ub`
ineqlin	Linear inequalities
eqlin	Linear equalities

See "Lagrange Multiplier Structures" on page 3-25.

More About

Tips

- For problems with no constraints, you can use \ (matrix left division). When you have no constraints, `lsqlin` returns `x = C\d`.
- Because the problem being solved is always convex, `lsqlin` finds 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`.
- The `trust-region-reflective` algorithm does not allow equal upper and lower bounds. Use another algorithm for this case.
- If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.
- You can solve some large structured problems, including those where the `C` matrix is too large to fit in memory, using the `trust-region-reflective` algorithm with

a Jacobian multiply function. For information, see `trust-region-reflective` Algorithm Options.

Algorithms

Trust-Region-Reflective Algorithm

When the problem given to `lsqlin` has *only* upper and lower bounds; that is, no linear inequalities or equalities are specified, and the matrix `C` has at least as many rows as columns, the default algorithm is `trust-region-reflective`. 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-Reflective Least Squares" on page 10-3, and in particular "Large Scale Linear Least Squares" on page 10-6.

Active-Set Algorithm

`lsqlin` uses the `active-set` algorithm when you specify it with `optimoptions`, or when you give linear inequalities or equalities. The algorithm 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 "`active-set` `quadprog` Algorithm" on page 9-11.

Interior-Point Algorithm

The `'interior-point'` algorithm is based on the `quadprog` `'interior-point-convex'` algorithm. See "Interior-Point Linear Least Squares" on page 10-7.

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.
- "Least-Squares (Model Fitting) Algorithms" on page 10-2

See Also

`\` | `lsqnonneg` | `optimtool` | `quadprog`

Introduced before R2006a

lsqnonlin

Solve nonlinear least-squares (nonlinear data-fitting) problems

Nonlinear least-squares solver

Solves nonlinear least-squares curve fitting problems of the form

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$

with optional lower and upper bounds *lb* and *ub* on the components of *x*.

x, *lb*, and *ub* can be vectors or matrices; see "Matrix Arguments" on page 2-30.

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

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}.$$

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,exitflag,output] = lsqnonlin( ___ )
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
lsqnonlin( ___ )
```

Description

`x = lsqnonlin(fun,x0)` starts at the point `x0` and finds a minimum of the sum of squares of the functions described in `fun`. The function `fun` should return a vector of values and not the sum of squares of the values. (The algorithm implicitly computes the sum of squares of the components of `fun(x)`.)

Note: "Passing Extra Parameters" on page 2-50 explains how to pass extra parameters to the vector function `fun(x)`, 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`. You can fix the solution component `x(i)` by specifying `lb(i) = ub(i)`.

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.

`x = lsqnonlin(fun,x0,lb,ub,options)` minimizes with the optimization options specified in `options`. Use `optimoptions` 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 14-251. Create the `problem` structure by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

`[x,resnorm] = lsqnonlin( ___ )`, for any input arguments, returns the value of the squared 2-norm of the residual at `x`: `sum(fun(x).^2)`.

`[x,resnorm,residual,exitflag,output] = lsqnonlin( ___ )` additionally returns the value of the residual `fun(x)` at the solution `x`, a value `exitflag` that describes the exit condition, and a structure `output` that contains information about the optimization process.

```
[x,resnorm,residual,exitflag,output,lambda,jacobian] =
lsqnonlin( ___ )
```
 additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`, and the Jacobian of `fun` at the solution `x`.

Examples

Fit a Simple Exponential

Fit a simple exponential decay curve to data.

Generate data from an exponential decay model plus noise. The model is

$$y = \exp(-1.3t) + \varepsilon,$$

with t ranging from 0 through 3, and ε normally distributed noise with mean 0 and standard deviation 0.05.

```
rng default % for reproducibility
d = linspace(0,3);
y = exp(-1.3*d) + 0.05*randn(size(d));
```

The problem is: given the data (`d`, `y`), find the exponential decay rate that best fits the data.

Create an anonymous function that takes a value of the exponential decay rate r and returns a vector of differences from the model with that decay rate and the data.

```
fun = @(r)exp(-d*r)-y;
```

Find the value of the optimal decay rate. Arbitrarily choose an initial guess `x0` = 4.

```
x0 = 4;
x = lsqnonlin(fun,x0)
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

```
x =

    1.2645
```

Plot the data and the best-fitting exponential curve.

```
plot(d,y,'ko',d,exp(-x*d),'b-')
legend('Data','Best fit')
xlabel('t')
ylabel('exp(-tx)')
```
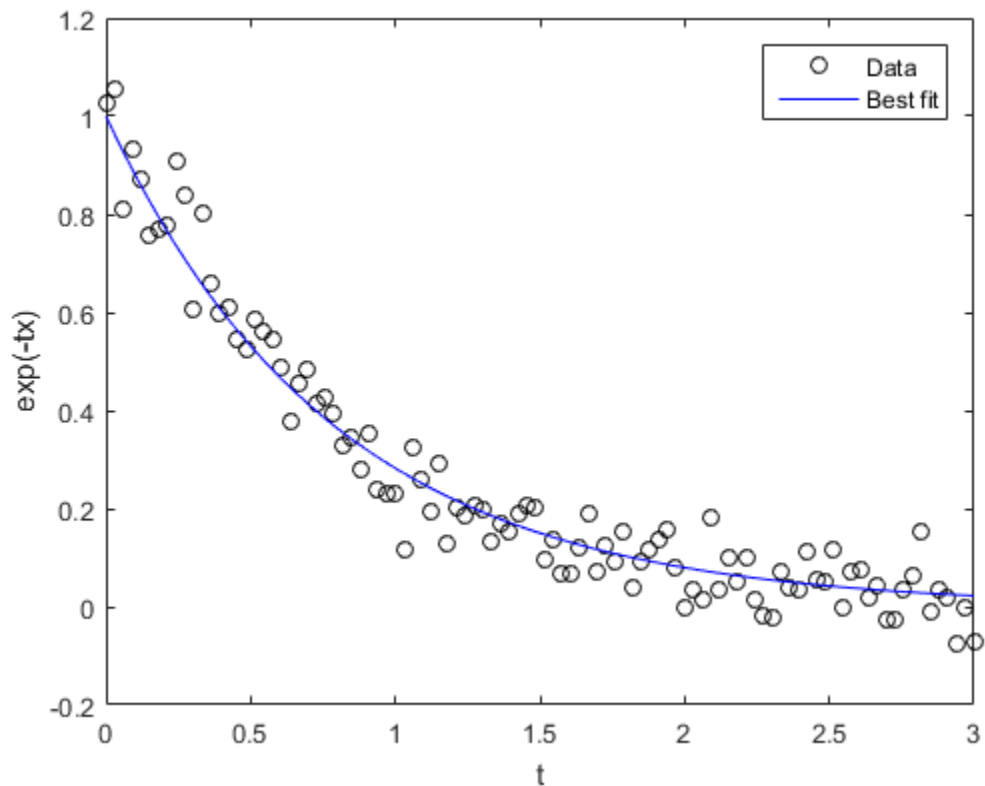

Fit a Problem with Bound Constraints

Find the best-fitting model when some of the fitting parameters have bounds.

Find a centering b and scaling a that best fit the function

$$a \exp(-t) \exp(- \exp(-(t - b)))$$

to the standard normal density,

$$\frac{1}{\sqrt{2\pi}} \exp\left(-t^2/2\right).$$

Create a vector `t` of data points, and the corresponding normal density at those points.

```
t = linspace(-4,4);
y = 1/sqrt(2*pi)*exp(-t.^2/2);
```

Create a function that evaluates the difference between the centered and scaled function from the normal `y`, with `x(1)` as the scaling a and `x(2)` as the centering b.

```
fun = @(x)x(1)*exp(-t).*exp(-exp(-(t-x(2)))) - y;
```

Find the optimal fit starting from `x0 = [1/2,0]`, with the scaling a between 1/2 and 3/2, and the centering b between -1 and 3.

```
lb = [1/2,-1];
ub = [3/2,3];
x0 = [1/2,0];
x = lsqnonlin(fun,x0,lb,ub)
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.



x =

    0.8231   -0.2444
```

Plot the two functions to see the quality of the fit.

```
plot(t,y,'r-',t,fun(x)+y,'b-')
xlabel('t')
legend('Normal density','Fitted function')
```
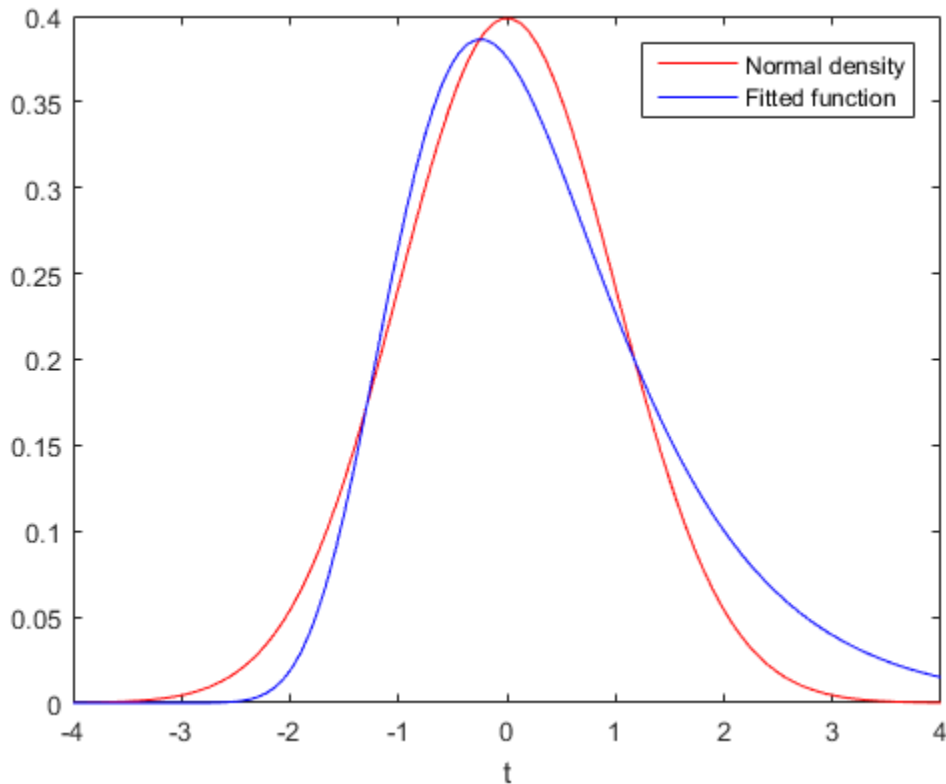

Nonlinear Least Squares with Nondefault Options

Compare the results of a data-fitting problem when using different `lsqnonlin` algorithms.

Suppose that you have observation time data `xdata` and observed response data `ydata`, and you want to find parameters $x(1)$ and $x(2)$ to fit a model of the form

$$\text{ydata} = x(1) \exp\left(x(2)\text{xdata}\right).$$

Input the observation times and responses.

```
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];
```

Create a simple exponential decay model. The model computes a vector of differences between predicted values and observed values.

```
fun = @(x)x(1)*exp(x(2)*xdata)-ydata;
```

Fit the model using the starting point `x0 = [100,-1]`. First, use the default `'trust-region-reflective'` algorithm.

```
x0 = [100,-1];
options = optimoptions(@lsqnonlin,'Algorithm','trust-region-reflective');
x = lsqnonlin(fun,x0,[],[],options)
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

```
x =

  498.8309   -0.1013
```

See if there is any difference using the `'levenberg-marquardt` algorithm.

```
options.Algorithm = 'levenberg-marquardt';
x = lsqnonlin(fun,x0,[],[],options)
```

```
Local minimum possible.

lsqnonlin stopped because the relative size of the current step is less than
the default value of the step size tolerance.
```

```
x =

  498.8309    -0.1013
```

The two algorithms found the same solution. Plot the solution and the data.

```
plot(xdata,ydata,'ko')
hold on
tlist = linspace(xdata(1),xdata(end));
plot(tlist,x(1)*exp(x(2)*tlist),'b-')
xlabel xdata
ylabel ydata
title('Exponential Fit to Data')
legend('Data','Exponential Fit')
hold off
```

Exponential Fit to Data
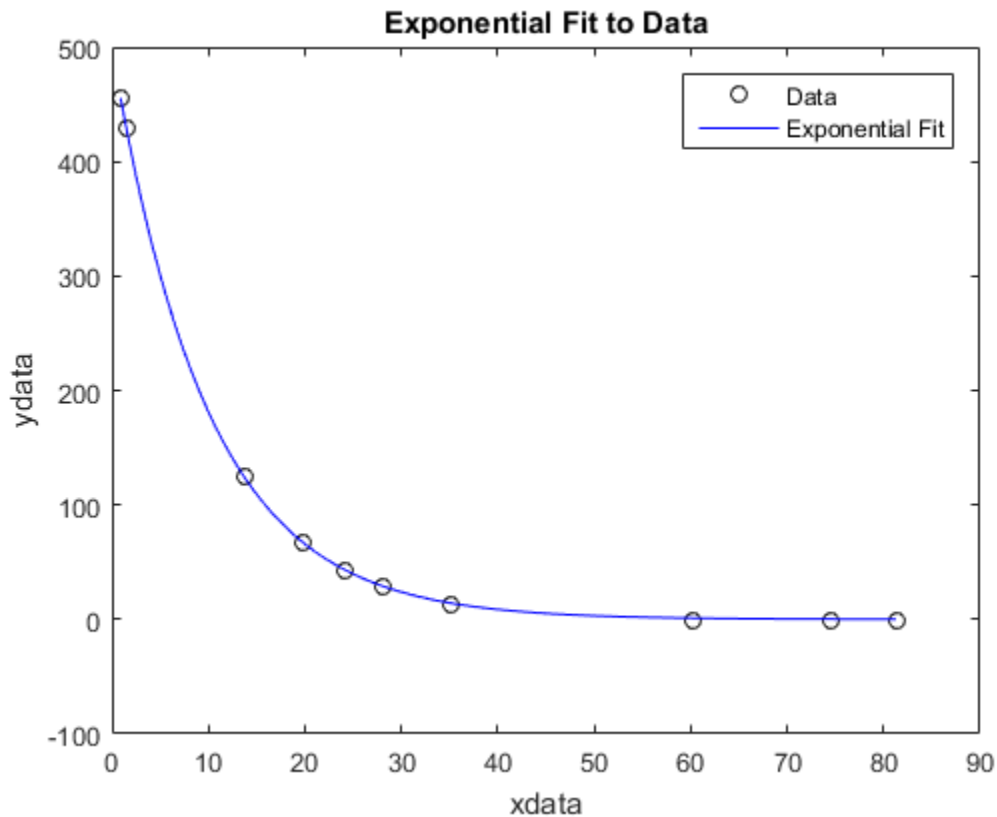
Nonlinear Least Squares Solution and Residual Norm

Find the *x* that minimizes

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

and find the value of the minimal sum of squares.

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 10 components).

First, write a file to compute the 10-component vector F.

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

Find the minimizing point and the minimum value, starting at the point x0 = [0.3,0.4].

```
x0 = [0.3,0.4];
[x,resnorm] = lsqnonlin(@myfun,x0);
```

After about 24 function evaluations, this example gives the solution

```
x,resnorm
```

```
x =
     0.2578    0.2578

resnorm =
     124.3622
```

Examine the Solution Process

Examine the solution process both as it occurs (by setting the Display option to 'iter') and afterwards (by examining the output structure).

Suppose that you have observation time data xdata and observed response data ydata, and you want to find parameters $x(1)$ and $x(2)$ to fit a model of the form

$$\text{ydata} = x(1) \exp\left(x(2)\text{xdata}\right).$$

Input the observation times and responses.

```
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];
```

Create a simple exponential decay model. The model computes a vector of differences between predicted values and observed values.

```
fun = @(x)x(1)*exp(x(2)*xdata)-ydata;
```

Fit the model using the starting point `x0 = [100,-1]`. Examine the solution process by setting the `Display` option to `'iter'`. Obtain an `output` structure to obtain more information about the solution process.

```
x0 = [100,-1];
options = optimoptions('lsqnonlin','Display','iter');
[x,resnorm,residual,exitflag,output] = lsqnonlin(fun,x0,[],[],options);
```

```
                                       Norm of      First-order
 Iteration  Func-count      f(x)         step        optimality
     0           3         359677                     2.88e+04
User objective function returned Inf; trying a new point...
     1           6         359677       11.6976       2.88e+04
     2           9         321395        0.5          4.97e+04
     3          12         321395        1            4.97e+04
     4          15         292253        0.25         7.06e+04
     5          18         292253        0.5          7.06e+04
     6          21         270350        0.125        1.15e+05
     7          24         270350        0.25         1.15e+05
     8          27         252777        0.0625       1.63e+05
     9          30         252777        0.125        1.63e+05
    10          33         243877        0.03125      7.48e+04
    11          36         243660        0.0625       8.7e+04
    12          39         243276        0.0625       2e+04
    13          42         243174        0.0625       1.14e+04
    14          45         242999        0.125        5.1e+03
    15          48         242661        0.25         2.04e+03
    16          51         241987        0.5          1.91e+03
    17          54         240643        1            1.04e+03
    18          57         237971        2            3.36e+03
    19          60         232686        4            6.04e+03
    20          63         222354        8            1.2e+04
    21          66         202592       16            2.25e+04
    22          69         166443       32            4.05e+04
    23          72         106320       64            6.68e+04
    24          75        28704.7      128            8.31e+04
    25          78        89.7947      140.674        2.22e+04
```

```
    26          81          9.57381         2.02599             684
    27          84          9.50489        0.0619927            2.27
    28          87          9.50489        0.00046226          0.0114

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the default value of the function tolerance.
```

Examine the output structure to obtain more information about the solution process.

```
output
```

```
output =

    firstorderopt: 0.0114
        iterations: 28
         funcCount: 87
       cgiterations: 0
         algorithm: 'trust-region-reflective'
          stepsize: 4.6226e-04
           message: 'Local minimum possible....'
```

For comparison, set the Algorithm option to `'levenberg-marquardt'`.

```
options.Algorithm = 'levenberg-marquardt';
[x,resnorm,residual,exitflag,output] = lsqnonlin(fun,x0,[],[],options);
```

```
                                           First-Order                     Norm of
 Iteration  Func-count    Residual        optimality      Lambda             step
     0          3          359677          2.88e+04         0.01
User objective function returned Inf; trying a new point...
     1         13          340761          3.91e+04        100000         0.280777
     2         16          304661          5.97e+04         10000         0.373146
     3         21          297292          6.55e+04         1e+06        0.0589933
     4         24          288240          7.57e+04        100000        0.0645444
     5         28          275407          1.01e+05         1e+06        0.0741266
     6         31          249954          1.62e+05        100000         0.094571
     7         36          245896          1.35e+05         1e+07        0.0133606
     8         39          243846          7.26e+04         1e+06        0.0094431
```

9	42	243568	5.66e+04	100000	0.0082162
10	45	243424	1.61e+04	10000	0.00777935
11	48	243322	8.8e+03	1000	0.0673933
12	51	242408	5.1e+03	100	0.675209
13	54	233628	1.05e+04	10	6.59804
14	57	169089	8.51e+04	1	54.6992
15	60	30814.7	1.54e+05	0.1	196.939
16	63	147.496	8e+03	0.01	129.795
17	66	9.51503	117	0.001	9.96069
18	69	9.50489	0.0714	0.0001	0.080486
19	72	9.50489	4.86e-05	1e-05	5.07037e-05

```
Local minimum possible.

lsqnonlin stopped because the relative size of the current step is less than
the default value of the step size tolerance.
```

The `'levenberg-marquardt'` converged with fewer iterations, but almost as many function evaluations:

output

```
output =

      iterations: 19
       funcCount: 72
        stepsize: 5.0704e-05
     cgiterations: []
   firstorderopt: 4.8597e-05
       algorithm: 'levenberg-marquardt'
         message: 'Local minimum possible....'
```

- "Nonlinear Least Squares (Curve Fitting)"

Input Arguments

fun — Function whose sum of squares is minimized
function handle | name of function

Function whose sum of squares is minimized, specified as a function handle or the name of a function. `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 to a file:

```
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.

Note The sum of squares should not be formed explicitly. Instead, your function should return a vector of function values. See "Examples" on page 14-   .

If the Jacobian can also be computed *and* the Jacobian option is `'on'`, set by

```
options = optimoptions('lsqnonlin','Jacobian','on')
```

then the function `fun` must return a second output argument with 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`.)

Example: `@(x)cos(x).*exp(-x)`

Data Types: `char` | `function_handle`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

lb — Lower bounds
real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `lb`, then `lb` specifies that
`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(x0)`, then `lb` specifies that
`x(i) >= lb(i)` for `1 <= i <= numel(lb)`.

In this case, solvers issue a warning.

Example: To specify that all x-components are positive, `lb = zeros(size(x0))`

Data Types: `double`

ub — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `ub`, then `ub` specifies that
`x(i) <= ub(i)` for all `i`.

If `numel(ub) < numel(x0)`, then `ub` specifies that
`x(i) <= ub(i)` for `1 <= i <= numel(ub)`.

In this case, solvers issue a warning.

Example: To specify that all x-components are less than one, `ub = ones(size(x0))`

Data Types: `double`

options — Optimization options
output of optimoptions | structure as optimset returns

Optimization options, specified as the output of optimoptions or a structure as optimset returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 13-7 for detailed information.

All Algorithms

Algorithm	Choose between 'trust-region-reflective' (default) and 'levenberg-marquardt'. Set the initial Levenberg-Marquardt parameter λ by setting Algorithm to a cell array such as {'levenberg-marquardt',.005}. The default $\lambda = 0.01$.
	The Algorithm option specifies a preference for which algorithm to use. It is only a preference, because certain conditions must be met to use each 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. The Levenberg-Marquardt algorithm does not handle bound constraints. For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-7.
DerivativeCheck	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are 'off' (default) or 'on'.
Diagnostics	Display diagnostic information about the function to be minimized or solved. Choices are 'off' (default) or 'on'.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.
Display	Level of display (see "Iterative Display" on page 3-16):
	• 'off' or 'none' displays no output.

- `'iter'` displays output at each iteration, and gives the default exit message.

- `'iter-detailed'` displays output at each iteration, and gives the technical exit message.

- `'final'` (default) displays just the final output, and gives the default exit message.

- `'final-detailed'` displays just the final output, and gives the technical exit message.

FinDiffRelStep	Scalar or vector step size factor for finite differences. When you set `FinDiffRelStep` to a vector `v`, forward finite differences steps `delta` are `delta = v.*sign′(x).*max(abs(x),TypicalX);` where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are `delta = v.*max(abs(x),TypicalX);` Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
FinDiffType	Finite differences, used to estimate gradients, are either `'forward'` (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 could take a backward, rather than a forward, difference to avoid evaluating at a point outside bounds.
FunValCheck	Check whether function values are valid. `'on'` displays an error when the function returns a value that is `complex`, `Inf`, or `NaN`. The default `'off'` displays no error.
Jacobian	If `'off'` (default), the solver approximates the Jacobian using finite differences. If `'on'`, the solver uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function.

MaxFunEvals	Maximum number of function evaluations allowed, a positive integer. The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
MaxIter	Maximum number of iterations allowed, a positive integer. The default is `400`. See "Tolerances and Stopping Criteria" on page 2-61 and "Iterations and Function Counts" on page 3-10.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[ ]`). See "Output Function" on page 13-20.
PlotFcns	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`):

- `@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 optimality measure.

	For information on writing a custom plot function, see "Plot Functions" on page 13-29.
TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TolX	Termination tolerance on `x`, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-61.
TypicalX	Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. The solver uses `TypicalX` for scaling finite differences for gradient estimation.

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. The solver uses `Jinfo` to compute the preconditioner. See "Passing Extra Parameters" on page 2-50 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'Jacobian'` must be set to `'on'` for the solver to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-75 and "Jacobian Multiply Function with Linear Least Squares" on page 10-26 for similar examples.

JacobPattern	Sparsity pattern of the Jacobian for finite differencing. Set `JacobPattern(i,j) = 1` when `fun(i)` depends on `x(j)`. Otherwise, set `JacobPattern(i,j) = 0`. In other words, `JacobPattern(i,j) = 1` when you can have ∂`fun(i)`/∂`x(j)` $\neq 0$.
	Use `JacobPattern` when it is inconvenient to compute the Jacobian matrix J in `fun`, though you can determine (say, by inspection) when `fun(i)` depends on `x(j)`. The solver can approximate J via sparse finite differences when you give `JacobPattern`.
	If the structure is unknown, do not set `JacobPattern`. The default behavior is as if `JacobPattern` is a dense matrix of ones. Then the solver computes a full finite-difference approximation in each iteration. This can be expensive for large problems, so it is usually better to determine the sparsity structure.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,numberOfVariables/2)`. For more information, see "Large Scale Nonlinear Least Squares" on page 10-5.
PrecondBandWidth	Upper bandwidth of preconditioner for PCG, a nonnegative integer. 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, a positive scalar. The default is `0.1`.

Levenberg-Marquardt Algorithm

InitDamping	Initial value of the Levenberg-Marquardt parameter, a positive scalar. Default is `1e-2`. For details, see "Levenberg-Marquardt Method" on page 10-7.

| ScaleProblem | 'Jacobian' can sometimes improve the convergence of a poorly scaled problem; the default is 'none'. |

Example: `options = optimoptions('lsqnonlin','Jacobian','on')`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
objective	Objective function
x0	Initial point for x
lb	Vector of lower bounds
ub	Vector of upper bounds
solver	'lsqnonlin'
options	Options created with optimoptions

You must supply at least the objective, x0, solver, and options fields in the problem structure.

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: struct

Output Arguments

x — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when exitflag is positive. For information on the quality of the solution, see "When the Solver Succeeds" on page 4-22.

`resnorm` — Norm of the residual
nonnegative real

Norm of the residual, returned as a nonnegative real. `resnorm` is the squared 2-norm of the residual at x: `sum(fun(x).^2)`.

`residual` — Value of objective function at solution
vector

Value of objective function at solution, returned as a vector. In general, `residual = fun(x)`.

`exitflag` — Reason the solver stopped
integer

Reason the solver stopped, returned as an integer.

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.MaxFunEvals`.
-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.

`output` — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

firstorderopt	Measure of first-order optimality
iterations	Number of iterations taken
funcCount	The number of function evaluations

cgiterations	Total number of PCG iterations (trust-region-reflective algorithm only)
stepsize	Final displacement in x
algorithm	Optimization algorithm used
message	Exit message

`lambda` — Lagrange multipliers at the solution
structure

Lagrange multipliers at the solution, returned as a structure with fields:

lower	Lower bounds lb
upper	Upper bounds ub

`jacobian` — Jacobian at the solution
real matrix

Jacobian at the solution, returned as a real matrix. `jacobian(i,j)` is the partial derivative of `fun(i)` with respect to `x(j)` at the solution `x`.

Limitations

- The Levenberg-Marquardt algorithm does not handle bound constraints.

- The trust-region-reflective algorithm 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, `lsqnonlin` uses the Levenberg-Marquardt algorithm.

  Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems that have both of these characteristics cannot be solved by `lsqnonlin`.

- `lsqnonlin` can solve complex-valued problems directly with the `levenberg-marquardt` algorithm. However, this algorithm does not accept bound constraints. For a complex problem with bound constraints, split the variables into real and imaginary parts, and use the `trust-region-reflective` algorithm. See "Fit a Model to Complex-Valued Data" on page 10-33.

- 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.

You can use the trust-region reflective algorithm in lsqnonlin, lsqcurvefit, and fsolve with small- to medium-scale problems without computing the Jacobian in fun or providing the Jacobian sparsity pattern. (This also applies to 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 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. You can find out only by running it and seeing if MATLAB runs within the amount of virtual memory available on your system.

More About

Algorithms

The Levenberg-Marquardt and trust-region-reflective methods are based on the nonlinear least-squares algorithms also used in fsolve.

- The default 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 Least Squares" on page 10-3.

- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 10-7.

- "Optimization Problem Setup"

- "Least-Squares (Model Fitting) Algorithms" on page 10-2

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, 1996, pp. 418–445.

[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, 1994, pp. 189–224.

[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*, 1944, pp. 164–168.

[5] Marquardt, D. "An Algorithm for Least-squares Estimation of Nonlinear Parameters." *SIAM Journal Applied Mathematics*, Vol. 11, 1963, pp. 431–441.

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

[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

`fsolve` | `lsqcurvefit` | `optimoptions`

Introduced before R2006a

lsqnonneg

Solve nonnegative least-squares constraint problem

Equation

Solves nonnegative least-squares curve fitting problems of the form

$$\min_x \left\| C \cdot x - d \right\|_2^2, \text{ where } x \geq 0.$$

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,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,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 14-265.

Create the structure `problem` by exporting a problem from Optimization app, as described in "Exporting Your Work" on page 5-11.

[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 d-C*x.

[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 13-2 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 Reference" on page 13-7 for detailed information.	
	Display	Level of display:
		• 'off' or 'none' 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, a positive scalar. The default is 10*eps*norm(C,1)*length(C). See "Tolerances and Stopping Criteria" on page 2-61.
problem	C	Matrix
	d	Vector
	solver	'lsqnonneg'
	options	Options structure created using optimset

Output Arguments

"Function Arguments" on page 13-2 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	'active-set'
	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)]
ans =
    -2.5627    0
     3.1108    0.6929

[norm(C*(C\d)-d), norm(C*lsqnonneg(C,d)-d)]
ans =
         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.

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.

More About

Algorithms

`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`.

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least-Squares Problems*, Prentice-Hall, Chapter 23, p. 161, 1974.

See Also
\ | lsqlin | optimset | optimtool

mpsread

Read MPS file for LP and MILP optimization data

Syntax

```
problem = mpsread(mpsfile)
```

Description

`problem = mpsread(mpsfile)` reads data for linear programming (LP) and mixed-integer linear programming (MILP) problems. It returns the data in a structure that the `intlinprog` or `linprog` solvers accept.

Note: `mpsread` does not support semicontinuous constraints or SOS constraints.

Examples

Import and Run an MPS File

Load an `mps` file and solve the problem it describes.

Load the `eil33.2.mps` file from a public repository. View the problem type.

```
gunzip('http://miplib.zib.de/download/eil33-2.mps.gz')
problem = mpsread('eil33-2.mps')

problem =

        f: [4516x1 double]
    Aineq: [0x4516 double]
    bineq: [0x1 double]
      Aeq: [32x4516 double]
      beq: [32x1 double]
       lb: [4516x1 double]
```

```
      ub: [4516x1 double]
  intcon: [4516x1 double]
  solver: 'intlinprog'
 options: [1x1 optim.options.Intlinprog]
```

Notice that `problem.intcon` is not empty, and `problem.solver` is `'intlinprog'`. The problem is an integer linear programming problem.

Change the options to suppress iterative display and to generate a plot as the solver progresses.

```
options = optimoptions('intlinprog','Display','final','PlotFcns',@optimplotmilp);
problem.options = options;
```

Solve the problem by calling `intlinprog`.

```
[x,fval,exitflag,output] = intlinprog(problem);
```

```
Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the optimal
value, options.TolGapRel = 0.0001 (the default value). The intcon variables are
integer within tolerance, options.TolInteger = 1e-05 (the default value).
```

Best objective: 934.008, Relative gap: 0.002618.
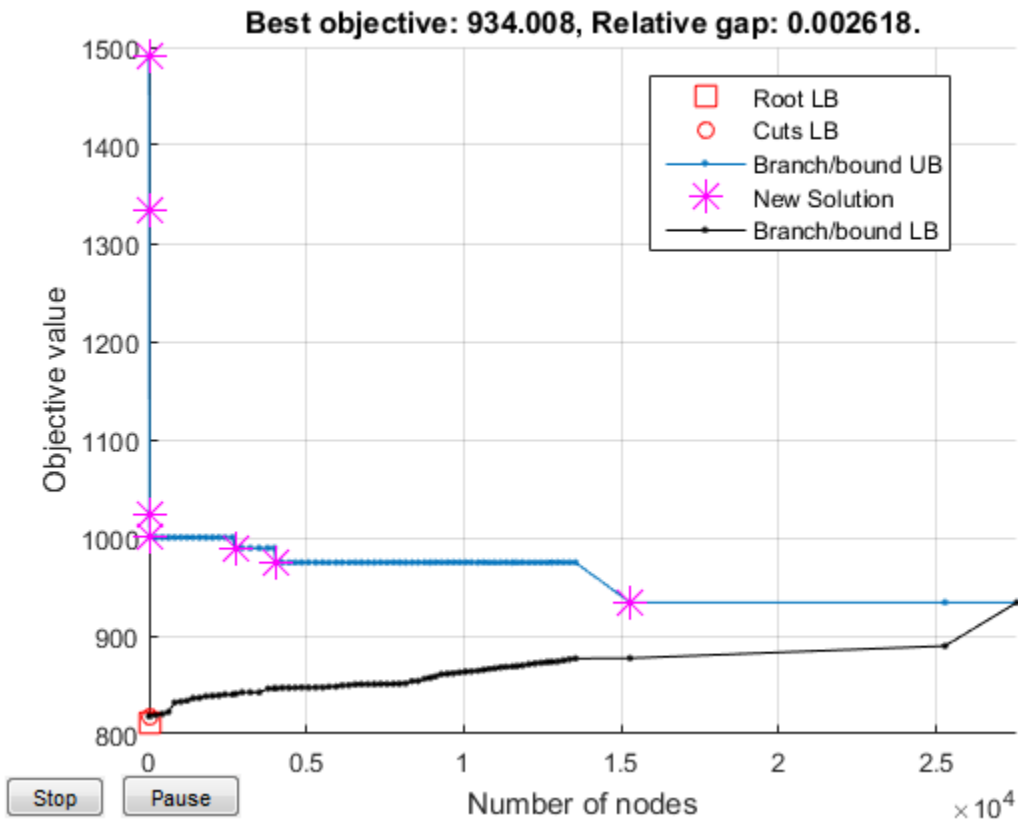
Input Arguments

mpsfile — Path to MPS file
string

Path to MPS file, specified as a string. mpsfile should be a file in the MPS format.

Example: 'documents/optimization/milpproblem.mps'

Data Types: char

Output Arguments

`problem` — Problem structure
structure

Problem structure, returned as a structure with fields:

`f`	Vector representing objective `f'*x`
`intcon`	Vector indicating variables that take integer values (empty for LP, nonempty for MILP)
`Aineq`	Matrix in linear inequality constraints `Aineq*x` $\leq$ `bineq`
`bineq`	Vector in linear inequality constraints `Aineq*x` $\leq$ `bineq`
`Aeq`	Matrix in linear equality constraints `Aeq*x = beq`
`beq`	Vector in linear equality constraints `Aeq*x = beq`
`lb`	Vector of lower bounds
`ub`	Vector of upper bounds
`solver`	`'intlinprog'` (if `intcon` is nonempty), or `'linprog'` (if `intcon` is empty)
`options`	Default options, as returned by the command `optimoptions(solver)`

`mpsread` returns `problem.Aineq` and `problem.Aeq` as sparse matrices.

More About

- "Linear Programming and Mixed-Integer Linear Programming"

See Also
`intlinprog` | `linprog`

Introduced in R2015b

optimget

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

optimoptions

Create optimization options

Syntax

```
options = optimoptions(SolverName)
options = optimoptions(SolverName,Name,Value)

options = optimoptions(oldoptions,Name,Value)

options = optimoptions(SolverName,oldoptions)
```

Description

`options = optimoptions(SolverName)` returns a set of default options for the `SolverName` solver.

`options = optimoptions(SolverName,Name,Value)` returns options with the named parameters altered with the specified values.

`options = optimoptions(oldoptions,Name,Value)` returns a copy of `oldoptions` with the named parameters altered with the specified values.

`options = optimoptions(SolverName,oldoptions)` returns default options for the `SolverName` solver, and copies the applicable options in `oldoptions` to `options`.

Examples

Create Default Options

Create default options for the `fmincon` solver.

```
options = optimoptions('fmincon')

options =

  fmincon options:
```

```
Options used by current Algorithm ('interior-point'):
(Other available algorithms: 'active-set', 'sqp', 'trust-region-reflective')

Set by user:
  No options set by user.

Default:
                 Algorithm: 'interior-point'
  AlwaysHonorConstraints: 'bounds'
         DerivativeCheck: 'off'
             Diagnostics: 'off'
           DiffMaxChange: Inf
           DiffMinChange: 0
                 Display: 'final'
          FinDiffRelStep: 'sqrt(eps)'
             FinDiffType: 'forward'
             FunValCheck: 'off'
              GradConstr: 'off'
                 GradObj: 'off'
                 HessFcn: []
                 Hessian: 'bfgs'
                 HessMult: []
        InitBarrierParam: 0.1000
   InitTrustRegionRadius: 'sqrt(numberOfVariables)'
              MaxFunEvals: 3000
                  MaxIter: 1000
             MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
           ObjectiveLimit: -1.0000e+20
                OutputFcn: []
                  PlotFcns: []
              ScaleProblem: 'none'
      SubproblemAlgorithm: 'ldl-factorization'
                    TolCon: 1.0000e-06
                    TolFun: 1.0000e-06
                 TolProjCG: 0.0100
               TolProjCGAbs: 1.0000e-10
                      TolX: 1.0000e-10
                  TypicalX: 'ones(numberOfVariables,1)'
               UseParallel: 0

Options not used by current Algorithm ('interior-point')
Default:
               HessPattern: 'sparse(ones(numberOfVariables))'
                 MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
```

```
                   MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
            PrecondBandWidth: 0
              RelLineSrchBnd: []
     RelLineSrchBndDuration: 1
                   TolConSQP: 1.0000e-06
                      TolPCG: 0.1000
```

Create Nondefault Options

Create nondefault options for the `fmincon` solver.

```
options = optimoptions(@fmincon,'Algorithm','sqp','MaxIter',1500)


options =

  fmincon options:

   Options used by current Algorithm ('sqp'):
   (Other available algorithms: 'active-set', 'interior-point', 'trust-region-reflectiv

   Set by user:
         Algorithm: 'sqp'
           MaxIter: 1500

   Default:
    DerivativeCheck: 'off'
        Diagnostics: 'off'
      DiffMaxChange: Inf
      DiffMinChange: 0
            Display: 'final'
      FinDiffRelStep: 'sqrt(eps)'
        FinDiffType: 'forward'
        FunValCheck: 'off'
         GradConstr: 'off'
            GradObj: 'off'
         MaxFunEvals: '100*numberOfVariables'
       ObjectiveLimit: -1.0000e+20
           OutputFcn: []
            PlotFcns: []
        ScaleProblem: 'none'
              TolCon: 1.0000e-06
              TolFun: 1.0000e-06
```

```
                   TolX: 1.0000e-06
               TypicalX: 'ones(numberOfVariables,1)'
            UseParallel: 0

   Options not used by current Algorithm ('sqp')
   Default:
    AlwaysHonorConstraints: 'bounds'
                   HessFcn: []
                  HessMult: []
               HessPattern: 'sparse(ones(numberOfVariables))'
                   Hessian: 'not applicable'
           InitBarrierParam: 0.1000
      InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                 MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
              MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
                  MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
           PrecondBandWidth: 0
             RelLineSrchBnd: []
     RelLineSrchBndDuration: 1
         SubproblemAlgorithm: 'ldl-factorization'
                  TolConSQP: 1.0000e-06
                     TolPCG: 0.1000
                   TolProjCG: 0.0100
                TolProjCGAbs: 1.0000e-10
```

Update Options

Update existing options with new values.

Create options for the lsqnonlin solver.

```
oldoptions = optimoptions(@lsqnonlin,'Algorithm','levenberg-marquardt','MaxFunEvals',15
```

```
oldoptions =

  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
   (Other available algorithms: 'trust-region-reflective')

   Set by user:
         Algorithm: 'levenberg-marquardt'
```

```
        MaxFunEvals: 1500

    Default:
     DerivativeCheck: 'off'
         Diagnostics: 'off'
       DiffMaxChange: Inf
       DiffMinChange: 0
             Display: 'final'
       FinDiffRelStep: 'sqrt(eps)'
          FinDiffType: 'forward'
          FunValCheck: 'off'
          InitDamping: 0.0100
             Jacobian: 'off'
              MaxIter: 400
            OutputFcn: []
             PlotFcns: []
         ScaleProblem: 'none'
               TolFun: 1.0000e-06
                 TolX: 1.0000e-06
              TypicalX: 'ones(numberOfVariables,1)'

    Options not used by current Algorithm ('levenberg-marquardt')
    Default:
            JacobMult: []
         JacobPattern: 'sparse(ones(Jrows,Jcols))'
            MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
     PrecondBandWidth: Inf
               TolPCG: 0.1000
```

Increase `MaxFunEvals` to 2000.

```
options = optimoptions(oldoptions,'MaxFunEvals',2000)


options =

  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
   (Other available algorithms: 'trust-region-reflective')

   Set by user:
         Algorithm: 'levenberg-marquardt'
```

```
         MaxFunEvals: 2000

    Default:
     DerivativeCheck: 'off'
         Diagnostics: 'off'
       DiffMaxChange: Inf
       DiffMinChange: 0
             Display: 'final'
       FinDiffRelStep: 'sqrt(eps)'
          FinDiffType: 'forward'
         FunValCheck: 'off'
          InitDamping: 0.0100
            Jacobian: 'off'
             MaxIter: 400
           OutputFcn: []
            PlotFcns: []
        ScaleProblem: 'none'
              TolFun: 1.0000e-06
                TolX: 1.0000e-06
            TypicalX: 'ones(numberOfVariables,1)'

   Options not used by current Algorithm ('levenberg-marquardt')
   Default:
            JacobMult: []
        JacobPattern: 'sparse(ones(Jrows,Jcols))'
           MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
     PrecondBandWidth: Inf
               TolPCG: 0.1000
```

Use Dot Notation to Update Options

Update existing options with new values by using dot notation.

Create options for the lsqnonlin solver.

```
options = optimoptions(@lsqnonlin,'Algorithm','levenberg-marquardt','MaxFunEvals',1500)

options =

  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
```

14-279

```
(Other available algorithms: 'trust-region-reflective')

Set by user:
      Algorithm: 'levenberg-marquardt'
     MaxFunEvals: 1500

Default:
 DerivativeCheck: 'off'
     Diagnostics: 'off'
   DiffMaxChange: Inf
   DiffMinChange: 0
         Display: 'final'
   FinDiffRelStep: 'sqrt(eps)'
     FinDiffType: 'forward'
     FunValCheck: 'off'
     InitDamping: 0.0100
        Jacobian: 'off'
         MaxIter: 400
       OutputFcn: []
        PlotFcns: []
    ScaleProblem: 'none'
          TolFun: 1.0000e-06
            TolX: 1.0000e-06
         TypicalX: 'ones(numberOfVariables,1)'

Options not used by current Algorithm ('levenberg-marquardt')
Default:
       JacobMult: []
     JacobPattern: 'sparse(ones(Jrows,Jcols))'
        MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
 PrecondBandWidth: Inf
           TolPCG: 0.1000
```

Increase `MaxFunEvals` to 2000 by using dot notation.

```
options.MaxFunEvals = 2000
```

```
options =

  lsqnonlin options:

  Options used by current Algorithm ('levenberg-marquardt'):
```

```
   (Other available algorithms: 'trust-region-reflective')

   Set by user:
         Algorithm: 'levenberg-marquardt'
        MaxFunEvals: 2000

   Default:
    DerivativeCheck: 'off'
        Diagnostics: 'off'
      DiffMaxChange: Inf
      DiffMinChange: 0
            Display: 'final'
      FinDiffRelStep: 'sqrt(eps)'
         FinDiffType: 'forward'
         FunValCheck: 'off'
         InitDamping: 0.0100
            Jacobian: 'off'
             MaxIter: 400
           OutputFcn: []
            PlotFcns: []
        ScaleProblem: 'none'
               TolFun: 1.0000e-06
                 TolX: 1.0000e-06
             TypicalX: 'ones(numberOfVariables,1)'

   Options not used by current Algorithm ('levenberg-marquardt')
   Default:
           JacobMult: []
        JacobPattern: 'sparse(ones(Jrows,Jcols))'
           MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
     PrecondBandWidth: Inf
               TolPCG: 0.1000
```

Copy Options to Another Solver

Transfer nondefault options for the `fmincon` solver to options for the `fminunc` solver.

Create nondefault options for the `fmincon` solver.

```
oldoptions = optimoptions(@fmincon,'Algorithm','sqp','MaxIter',1500)


oldoptions =
```

```
fmincon options:

 Options used by current Algorithm ('sqp'):
 (Other available algorithms: 'active-set', 'interior-point', 'trust-region-reflecti

 Set by user:
        Algorithm: 'sqp'
          MaxIter: 1500

 Default:
  DerivativeCheck: 'off'
      Diagnostics: 'off'
    DiffMaxChange: Inf
    DiffMinChange: 0
          Display: 'final'
   FinDiffRelStep: 'sqrt(eps)'
       FinDiffType: 'forward'
      FunValCheck: 'off'
        GradConstr: 'off'
           GradObj: 'off'
       MaxFunEvals: '100*numberOfVariables'
    ObjectiveLimit: -1.0000e+20
         OutputFcn: []
           PlotFcns: []
      ScaleProblem: 'none'
            TolCon: 1.0000e-06
            TolFun: 1.0000e-06
               TolX: 1.0000e-06
          TypicalX: 'ones(numberOfVariables,1)'
        UseParallel: 0

 Options not used by current Algorithm ('sqp')
 Default:
  AlwaysHonorConstraints: 'bounds'
                  HessFcn: []
                 HessMult: []
              HessPattern: 'sparse(ones(numberOfVariables))'
                  Hessian: 'not applicable'
          InitBarrierParam: 0.1000
    InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
             MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
                MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
```

```
          PrecondBandWidth: 0
             RelLineSrchBnd: []
    RelLineSrchBndDuration: 1
       SubproblemAlgorithm: 'ldl-factorization'
                 TolConSQP: 1.0000e-06
                    TolPCG: 0.1000
                  TolProjCG: 0.0100
               TolProjCGAbs: 1.0000e-10
```

Transfer the applicable options to the `fminunc` solver.

```
options = optimoptions(@fminunc,oldoptions)


options =

  fminunc options:

   Options used by current Algorithm ('trust-region'):
   (Other available algorithms: 'quasi-newton')

   Set by user:
             MaxIter: 1500

   Default:
           Algorithm: 'trust-region'
      DerivativeCheck: 'off'
         Diagnostics: 'off'
       DiffMaxChange: Inf
       DiffMinChange: 0
             Display: 'final'
       FinDiffRelStep: 'sqrt(eps)'
          FinDiffType: 'forward'
          FunValCheck: 'off'
              GradObj: 'off'
              Hessian: 'off'
             HessMult: []
          HessPattern: 'sparse(ones(numberOfVariables))'
          MaxFunEvals: '100*numberOfVariables'
            MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
             OutputFcn: []
              PlotFcns: []
     PrecondBandWidth: 0
```

```
            TolFun: 1.0000e-06
            TolPCG: 0.1000
              TolX: 1.0000e-06
          TypicalX: 'ones(numberOfVariables,1)'

Options not used by current Algorithm ('trust-region')
Default:
      HessUpdate: 'bfgs'
  ObjectiveLimit: -1.0000e+20
```

Input Arguments

`SolverName` — Solver name
string or function handle

Solver name, specified as a string or function handle.

Example: `'fmincon'`

Example: `@fmincon`

Data Types: `char` | `function_handle`

`oldoptions` — Options
options created using `optimoptions`

Options, specified as an options object. The `optimoptions` function creates options objects.

Example: `oldoptions = optimoptions(@fminunc)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `optimoptions(@fmincon,'Display','iter','TolFun',1e-10)` sets `fmincon` options to have iterative display, and to have a `TolFun` tolerance of `1e-10`.

For relevant name-value pairs, consult the options table for your solver:

- `fgoalattain` options
- `fmincon` options
- `fminimax` options
- `fminunc` options
- `fseminf` options
- `fsolve` options
- `intlinprog` options
- `linprog` options
- `lsqcurvefit` options
- `lsqlin` options
- `lsqnonlin` options
- `particleswarm` options (in Global Optimization Toolbox)
- `quadprog` options

Output Arguments

`options` — Options object
options object

Options object, returned as the options for the `SolverName` solver.

Alternative Functionality

App

You can set and modify options using the Optimization app (`optimtool`).

Note: The Optimization app warns that it will be removed in a future release.

More About

- "Set Options"

- "Optimization App" on page 5-2

See Also
optimset | optimtool

Introduced in R2013a

optimset

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, including their default values, see the reference pages for the optimization functions. "Optimization Options Reference" on page 13-7 provides descriptions of optimization options and which functions use them.

Use the command `optimset(@`*`solver`*`)` or the equivalent `optimset` *`solver`* to see the default values of relevant optimization options for a solver. Some solvers do not have a default value, since the default depends on the algorithm. For example, the default value of the `maxIter` option in the `fmincon` solver is 400 for the trust-region-reflective algorithm, but is 1000 for the interior-point algorithm.

You can also see the default values of all relevant options in the Optimization app. To see the options:

1   Start the Optimization app, e.g., with the `optimtool` command.
2   Choose the solver from the **Solver** menu.
3   Choose the algorithm, if applicable, from the **Algorithm** menu.
4   Read off the default values within the **Options** pane.

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 | optimoptions | optimtool

optimtool

Select solver and optimization options, run problems

`optimtool` warns that it will be removed in a future release.

Syntax

```
optimtool
optimtool(optstruct)
optimtool('solver')
```

Description

`optimtool` opens the Optimization app. Use the Optimization app to select a solver, optimization options, and run problems. See "Optimization App" on page 5-2 for a complete description of the Optimization app.

The Optimization app can be used to run any Optimization Toolbox solver except `intlinprog`, and any Global Optimization Toolbox solver except `GlobalSearch` and `MultiStart`. Results can be exported to a file or to the MATLAB workspace as a structure.

`optimtool(optstruct)` starts the Optimization app and loads `optstruct`. `optstruct` can be either optimization options or an optimization problem structure. Create optimization options with the `optimoptions` or `optimset` function, or by using the export option from the Optimization app. Create a problem structure by exporting the problem from the Optimization app to the MATLAB workspace. If you have Global Optimization Toolbox, you can create a problem structure for `fmincon`, `fminunc`, `lsqnonlin`, or `lsqcurvefit` using the `createOptimProblem` function.

`optimtool('solver')` starts the Optimization app with the specified solver, identified as a string, and the corresponding default options and problem fields. All Optimization Toolbox and Global Optimization Toolbox solvers are valid inputs to the `optimtool` function, except for `intlinprog`, `GlobalSearch`, and `MultiStart`.
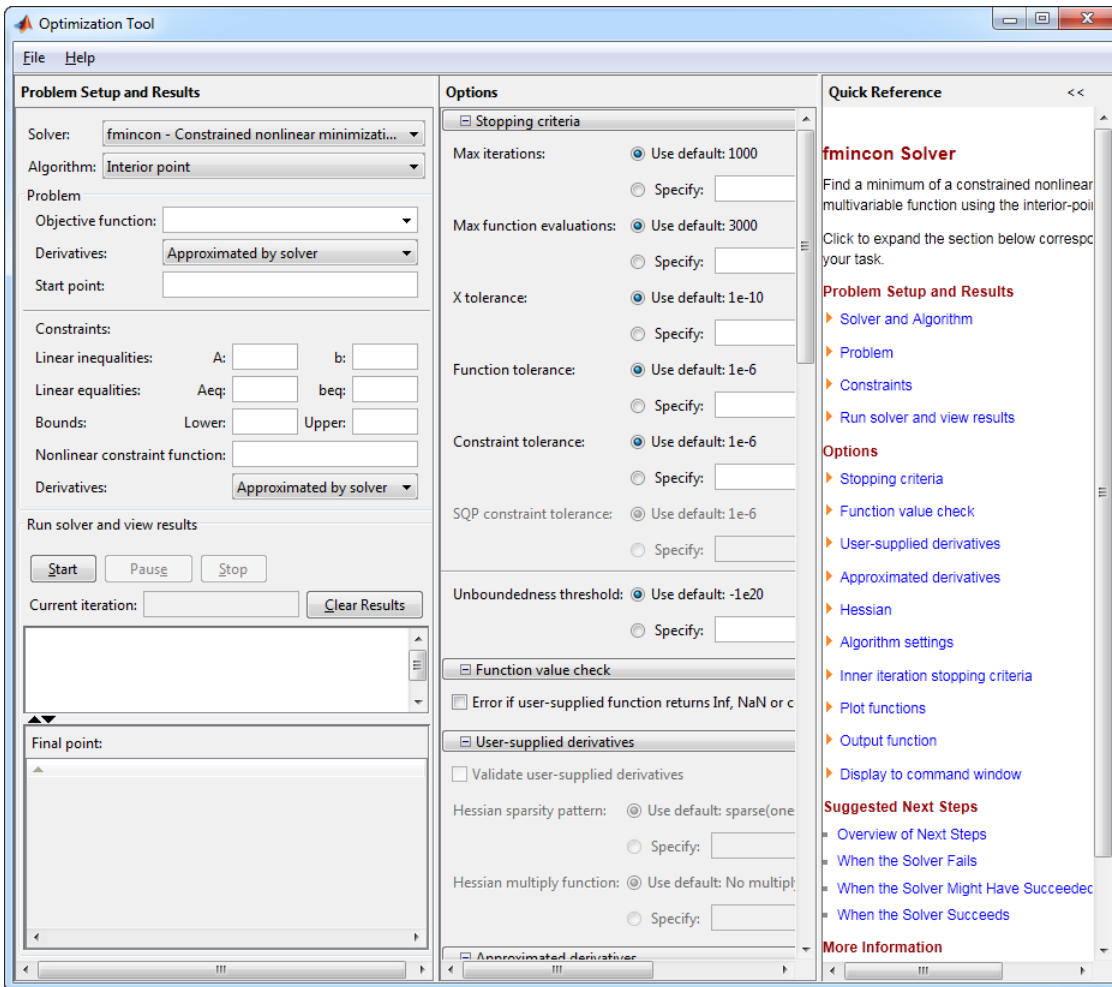
More About

- "Optimization App" on page 5-2
- "Solve a Constrained Nonlinear Problem" on page 1-3

See Also
optimoptions | optimset

quadprog

Quadratic programming

Syntax

```
x = quadprog(H,f)
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(H,f,...)
[x,fval,exitflag] = quadprog(H,f,...)
[x,fval,exitflag,output] = quadprog(H,f,...)
[x,fval,exitflag,output,lambda] = quadprog(H,f,...)
```

Description

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.

f, *lb*, and *ub* can be passed as vectors or matrices; see "Matrix Arguments" on page 2-30.

`x = quadprog(H,f)` returns a vector `x` that minimizes `1/2*x'*H*x + f'*x`. `H` must be positive definite for the problem to have a finite minimum.

`x = quadprog(H,f,A,b)` minimizes `1/2*x'*H*x + f'*x` subject to the restrictions `A*x` $\le$ `b`. `A` is a matrix of doubles, and `b` is a vector of doubles.

`x = quadprog(H,f,A,b,Aeq,beq)` solves the preceding problem subject to the additional restrictions `Aeq*x = beq`. `Aeq` is a matrix of doubles, and `beq` is a vector of doubles. If no inequalities exist, set `A = []` and `b = []`.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub)` solves the preceding problem subject to the additional restrictions $lb \leq x \leq ub$. `lb` and `ub` are vectors of doubles, and the restrictions hold for each `x` component. If no equalities exist, set `Aeq = []` and `beq = []`.

Note: If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

`quadprog` resets components of `x0` that violate the bounds $lb \leq x \leq ub$ to the interior of the box defined by the bounds. `quadprog` does not change components that respect the bounds.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0)` solves the preceding problem starting from the vector `x0`. If no bounds exist, set `lb = []` and `ub = []`. Some `quadprog` algorithms ignore `x0`, see "Input Arguments" on page 14-295.

`x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options)` solves the preceding problem using the optimization options specified in `options`. Use `optimoptions` to create `options`. If you do not want to give an initial point, set `x0 = []`.

`x = quadprog(problem)` returns the minimum for `problem`, where `problem` is a structure described in "Input Arguments" on page 14-295. Create `problem` by exporting a problem using the Optimization app; see "Exporting Your Work" on page 5-11.

`[x,fval] = quadprog(H,f,...)` returns the value of the objective function at `x`:

`fval = 0.5*x'*H*x + f'*x`

`[x,fval,exitflag] = quadprog(H,f,...)` `exitflag`, a scalar that describes the exit condition of `quadprog`.

`[x,fval,exitflag,output] = quadprog(H,f,...)` `output`, a structure that contains information about the optimization.

`[x,fval,exitflag,output,lambda] = quadprog(H,f,...)` `lambda`, a structure whose fields contain the Lagrange multipliers at the solution `x`.

Input Arguments

H

Symmetric matrix of doubles. Represents the quadratic in the expression `1/2*x'*H*x + f'*x`.

f

Vector of doubles. Represents the linear term in the expression `1/2*x'*H*x + f'*x`.

A

Matrix of doubles. Represents the linear coefficients in the constraints `A*x` $\leq$ `b`.

b

Vector of doubles. Represents the constant vector in the constraints `A*x` $\leq$ `b`.

Aeq

Matrix of doubles. Represents the linear coefficients in the constraints `Aeq*x = beq`.

beq

Vector of doubles. Represents the constant vector in the constraints `Aeq*x = beq`.

lb

Vector of doubles. Represents the lower bounds elementwise in `lb` $\leq$ `x` $\leq$ `ub`.

ub

Vector of doubles. Represents the upper bounds elementwise in `lb` $\leq$ `x` $\leq$ `ub`.

x0

Vector of doubles. Optional. The initial point for some `quadprog` algorithms:

- `active-set`

- `trust-region-reflective` when there are only bound constraints

If you do not give `x0`, `quadprog` sets all components of `x0` to a point in the interior of the box defined by the bounds. `quadprog` ignores `x0` for the `interior-point-convex` algorithm, and for the `trust-region-reflective` algorithm with equality constraints.

options

Options created using `optimoptions` or the Optimization app.

All Algorithms

`Algorithm`	Choose the algorithm:
	• `'interior-point-convex'` (default)
	• `'trust-region-reflective'`
	• `'active-set'` (will be removed in a future release)
	The `'trust-region-reflective'` algorithm handles problems with only bounds, or only linear equality constraints, but not both. The `'interior-point-convex'` algorithm handles only convex problems. For details, see "Choosing the Algorithm" on page 2-7.
`Diagnostics`	Display diagnostic information about the function to be minimized or solved. The choices are `'on'` or `'off'` (default).
`Display`	Level of display (see "Iterative Display" on page 3-16):
	• `'off'` or `'none'` displays no output.
	• `'final'` displays just the final output (default).
	The `'interior-point-convex'` algorithm allows additional values:
	• `'iter'` gives iterative display.
	• `'iter-detailed'` gives iterative display with a detailed exit message.
	• `'final-detailed'` displays just the final output, with a detailed exit message.
`MaxIter`	Maximum number of iterations allowed, a positive integer.

- For a `'trust-region-reflective'` equality-constrained problem, the default value is `2*(numberOfVariables - numberOfEqualities)`.

- For all other algorithms and problems, the default value is `200`.

All Algorithms Except active-set

TolFun   Termination tolerance on the function value, a positive scalar.

- For a `'trust-region-reflective'` equality-constrained problem, the default value is `1e-6`.

- For a `'trust-region-reflective'` bound-constrained problem, the default value is `100*eps`, about `2.2204e-14`.

- For `'interior-point-convex'`, the default value is `1e-8`.

TolX   Termination tolerance on `x`, a positive scalar.

- For `'trust-region-reflective'`, the default value is `100*eps`, about `2.2204e-14`.

- For `'interior-point-convex'`, the default value is `1e-8`.

trust-region-reflective Algorithm Only

HessMult   Function handle for a Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product `H*Y` without actually forming `H`. The function has the form

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

where `Hinfo` and possibly some additional parameters contain the matrices used to compute `H*Y`.

See "Quadratic Minimization with Dense, Structured Hessian" on page 9-19 for an example that uses this option.

MaxPCGIter   Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,floor(numberOfVariables/2))`. For more information, see "Preconditioned Conjugate Gradient Method" on page 9-9.

PrecondBandWidth	Upper bandwidth of the preconditioner for PCG, a nonnegative integer. By default, quadprog uses diagonal preconditioning (upper bandwidth 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 toward the solution.
TolPCG	Termination tolerance on the PCG iteration, a positive scalar. The default is 0.1.
TypicalX	Typical x values. The number of elements in TypicalX equals the number of elements in x0, the starting point. The default value is ones(numberofvariables,1). quadprog uses TypicalX internally for scaling. TypicalX has an effect only when x has unbounded components, and when a TypicalX value for an unbounded component exceeds 1.

interior-point-convex Algorithm Only

TolCon	Tolerance on the constraint violation, a positive scalar. The default is 1e-8.

problem

Structure encapsulating the quadprog inputs and options:

H	Symmetric matrix in 1/2*x'*H*x
f	Vector in linear term f'*x
Aineq	Matrix in linear inequality constraints Aineq*x $\leq$ bineq
bineq	Vector in linear inequality constraints Aineq*x $\leq$ bineq
Aeq	Matrix in linear equality constraints Aeq*x = beq
beq	Vector in linear equality constraints Aeq*x = beq
lb	Vector of lower bounds
ub	Vector of upper bounds
x0	Initial point for x

| solver | `'quadprog'` |
| options | Options created using `optimoptions` or the Optimization app |

Output Arguments

x

Vector that minimizes `1/2*x'*H*x + f'*x` subject to all bounds and linear constraints. `x` can be a local minimum for nonconvex problems. For convex problems, `x` is a global minimum. For more information, see "Local vs. Global Optima" on page 4-26.

fval

Value of `1/2*x'*H*x + f'*x` at the solution `x`, a double.

exitflag

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

All Algorithms

1	Function converged to the solution `x`.
0	Number of iterations exceeded `options.MaxIter`.
-2	Problem is infeasible.
-3	Problem is unbounded.

interior-point-convex Algorithm

| -6 | Nonconvex problem detected. |

trust-region-reflective Algorithm

| 3 | Change in the objective function value was smaller than `options.TolFun`. |
| -4 | Current search direction was not a direction of descent. No further progress could be made. |

active-set Algorithm

| 4 | Local minimizer was found. |

`-7`	Magnitude of search direction became too small. No further progress could be made. The problem is ill-posed or badly conditioned.

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 (trust-region-reflective algorithm only)
`constrviolation`	Maximum of constraint functions
`firstorderopt`	Measure of first-order optimality
`message`	Exit message

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

For details, see "Lagrange Multiplier Structures" on page 3-25.

Examples

Solve a simple quadratic programming problem: 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

$$x_1 + x_2 \leq 2$$
$$-x_1 + 2x_2 \leq 2$$
$$2x_1 + x_2 \leq 3$$
$$0 \leq x_1, \ 0 \leq x_2.$$

In matrix notation this is

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

where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \ f = \begin{bmatrix} -2 \\ -6 \end{bmatrix}, \ x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

1 Enter the 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);
```

2 Set the options to use the `'interior-point-convex'` algorithm with no display:

```
options = optimoptions('quadprog',...
    'Algorithm','interior-point-convex','Display','off');
```

3 Call quadprog:

```
[x,fval,exitflag,output,lambda] = ...
    quadprog(H,f,A,b,[],[],lb,[],[],options);
```

4 Examine the final point, function value, and exit flag:

```
 x,fval,exitflag

x =
    0.6667
    1.3333

fval =
    -8.2222
```

```
exitflag =
      1
```

5 An exit flag of `1` means the result is a local minimum. Because `H` is a positive definite matrix, this problem is convex, so the minimum is a global minimum. You can see `H` is positive definite by noting all its eigenvalues are positive:

```
eig(H)
ans =
    0.3820
    2.6180
```

Use the `'interior-point-convex'` algorithm to solve a sparse quadratic program.

1 Generate a sparse symmetric matrix for the quadratic form:

```
v = sparse([1,-.25,0,0,0,0,0,-.25]);
H = gallery('circul',v);
```

2 Include the linear term for the problem:

```
f = -4:3;
```

3 Include the constraint that the sum of the terms in the solution `x` must be less than `-2`:

```
A = ones(1,8);b = -2;
```

4 Set options to use the `'interior-point-convex'` algorithm and iterative display:

```
opts = optimoptions('quadprog',...
    'Algorithm','interior-point-convex','Display','iter');
```

5 Run the `quadprog` solver and observe the iterations:

```
[x fval eflag output lambda] = quadprog(H,f,A,b,[],[],[],[],[],opts);

                               First-order   Total relative
  Iter        f(x)     Feasibility    optimality        error
     0  -2.000000e+000   1.000e+001   4.500e+000   1.200e+001
     1  -2.630486e+001   0.000e+000   9.465e-002   9.465e-002
     2  -2.639877e+001   0.000e+000   3.914e-005   3.914e-005
     3  -2.639881e+001   0.000e+000   3.069e-015   6.883e-015

Minimum found that satisfies the constraints.

Optimization completed because the objective function is
```

non-decreasing in feasible directions, to within the default value
of the function tolerance, and constraints are satisfied to within
the default value of the constraint tolerance.

6 Examine the solution:

```
fval,eflag

fval =
  -26.3988

eflag =
     1
```

For the `'interior-point-convex'` algorithm, an exit flag of 1 means the result is
a global minimum.

Alternatives

You can use the Optimization app for quadratic programming. Enter `optimtool` at
the MATLAB command line, and choose the `quadprog - Quadratic programming`
solver. For more information, see "Optimization App" on page 5-2.

More About

Algorithms

interior-point-convex

The `'interior-point-convex'` algorithm attempts to follow a path that is strictly
inside the constraints. It uses a presolve module to remove redundancies, and to simplify
the problem by solving for components that are straightforward. For more information,
see "`interior-point-convex quadprog` Algorithm" on page 9-2.

trust-region-reflective

The `'trust-region-reflective'` 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). For more information, see "`trust-region-reflective quadprog` Algorithm" on page 9-6.

active-set

`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. For more information, see "`active-set quadprog` Algorithm" on page 9-11.

· "Optimization Problem Setup"

· "Optimization Results"

· "Quadratic Programming"

· "Mixed-Integer Quadratic Programming Portfolio Optimization" on page 8-85

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.

[3] Gould, N. and P. L. Toint. "Preprocessing for quadratic programming." *Math. Programming*, Series B, Vol. 100, pp. 95–132, 2004.

See Also
`linprog | lsqlin | optimoptions`